Getting Started

Release: NSO 6.2.5

Published: May 17, 2010

Last Modified: May 14, 2024

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1

About

Purpose

This guide helps you to develop a foundational understanding of NSO's core concepts and the installation process. The topics covered in this guide are intended to be introductory in nature to help you get started with NSO at a beginner level. Consult the other guides for advanced topics.

Target Audience

The guide is intended for evaluators, developers, system administrators, and other users who want to understand and install NSO.

About NSO

Cisco Network Service Orchestrator (NSO) is an evolution of the Tail-f Network Control System (NCS). Cisco acquired Tail-f in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the terms 'ncs' and 'tail-f' are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document, the term NSO is used to refer to the product, which consists of a number of tools and executables. These executable components will be referred to by their command-line name, e.g. ncs, ncs-netsim, ncs_cli, etc.

CHAPTER 2

Introduction to NSO

NSO Overview

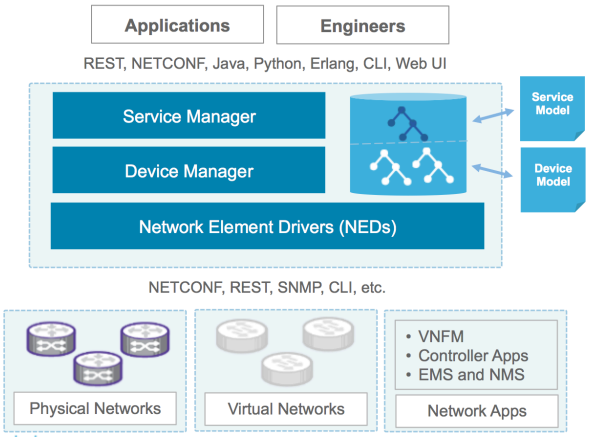
Cisco Network Services Orchestrator (NSO) enabled by Tail-f is an industry-leading orchestration platform for hybrid networks. It provides comprehensive lifecycle service automation to enable you to design and deliver high-quality services faster and more easily.

Creating and configuring network services is a complex task that often requires multiple configuration changes to all devices participating in the service. Additionally changes generally need to be made concurrently across all devices with the changes being either completely successful or rolled back to the starting configuration. And configuration need to be kept in sync across the system and the network devices. NSO approaches these challenges by acting as interface between people or software that want to configure the network, and the devices in the network.

The key features of NSO that comes into play includes:

1. Multi-vendor device configuration management using the native protocols of the network devices.
2. A Configuration Database (CDB) managing synchronized configurations for all devices and services in the network domain.
3. A set of northbound interfaces including human interfaces like web UI and a CLI; programmable interfaces including RESTCONF, NETCONF, JSON-RPC; and language bindings including Java, Python and Erlang.

Figure 1. NSO Overview



Network engineers use NSO as a central point of access to manage entire networks. They commonly do this using the NSO CLI or web UI. This document illustrates the use cases using CLI examples, but it is important to understand that any northbound interface can be used to achieve the same functionality.

All devices and services in the network can be accessed and configured using the NSO CLI, making it a powerful tool for network engineers. The CLI also provides an easy way to define roles and associated authorization policies limiting the engineers view of the devices under NSO control. Policies and integrity constraints can also be defined making sure the configuration adhere to operator standards.

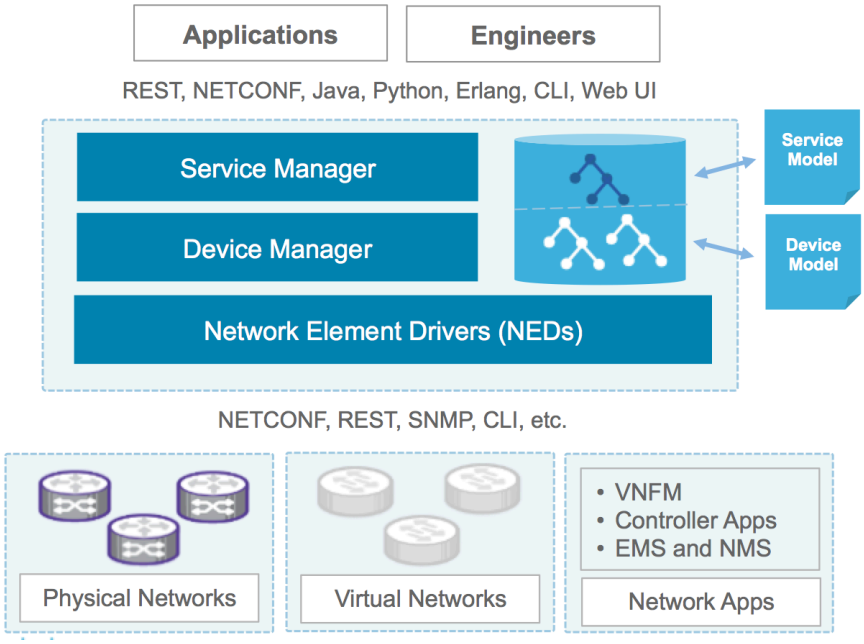
The typical workflow when using the NSO CLI is as follows:

1. The user logs in to the CLI and thereby starts a new session. The session provides a (logical) copy of the running configuration of CDB as a scratch pad for edits.
2. Changes made to the scratch pad are optionally validated at any time against policies and schemas using the "validate" command. Changes can always be viewed and verified prior to committing them.
3. The changes are committed, meaning that the changes are copied to the NSO database and deltas are pushed out to the network devices that are affected by the change. Changes that violate integrity constraints or network policies will not be committed but produce validation errors. The changes to the devices are done in a distributed and atomic transaction across all devices in parallel.
4. Changes either succeed and remain committed to device configuration or fail and are rolled back as a whole returning the entire network to the prior state.

Architecture Overview

This section provides a broad overview of the NSO architecture and functionality.

Figure 2. NSO Architecture



NSO has two main layers: the *Device Manager* and the *Service Manager*. They serve different purposes but are tightly integrated with a transactional engine and database.

The purpose of the Device Manager is to manage device configurations in a transactional manner. It supports features like fine-grained configuration commands, bidirectional device configuration synchronization, device groups and templates, and compliance reporting.

The Service Manager makes it possible for an operator to manage high-level aspects of the network that are not supported by the devices directly, or is supported in a cumbersome way. With the appropriate service definition running in the Service Manager, an operator could for example configure the VLANs that should exist in the network in a single place, and the Service Manager compute the specific configuration changes required for each device in the network and push them out. This covers the whole life-cycle for a service: creation, modification and deletion. NSO has an intelligent and easy way to use a mapping layer so that network engineers can define how a service should be deployed in the network.

NSO uses a dedicated built-in storage Configuration Database (CDB) for all configuration data. NSO keeps the CDB in sync with the real network device configurations. Audit, to ensure configuration consistency, and reconciliation, to synchronize configuration with the devices, functions are supported. It also maintains the runtime relationships between service instances and the corresponding device configurations.

NSO uses Network Element Drivers, NEDs, to communicate with devices. NEDs are not closed hard-coded adapters. Rather, the device interface is modeled in a data-model using the YANG data modelling language. NSO can render the required commands or operations directly from this model. This includes support for legacy configuration interfaces like device CLIs. This means that the NEDs can easily be updated to support new commands just by extending the data-models with the appropriate model constructs which avoids any programming tasks as part of the change cycle.

NSO also comes with tooling for simulating the configuration aspects of a network. The *netsim* tool is used to simulate management interfaces like Cisco CLI and NETCONF for the purpose of examples and service development.

The Orchestration Challenge

The industry is rapidly moving towards a service-oriented approach to network management, where complex services are supported by multi-vendor devices, physical and virtual. To manage these, operators are starting a transition from manually managing devices towards a situation where an operator is actively managing the various aspects of *services*.

Configuring the services and the affected devices are among the largest cost-drivers in provider networks. Still, the common orchestration and configuration management practice involves pervasive manual work or ad hoc scripting. Why do we still apply these sorts of techniques to the configuration management problem? Two of the primary reasons are the *variations of services* and the constant *change of devices*. These two underlying characteristics are, to some degree, blocking automated solutions, since it takes too long to update the solution to cope with daily changes.

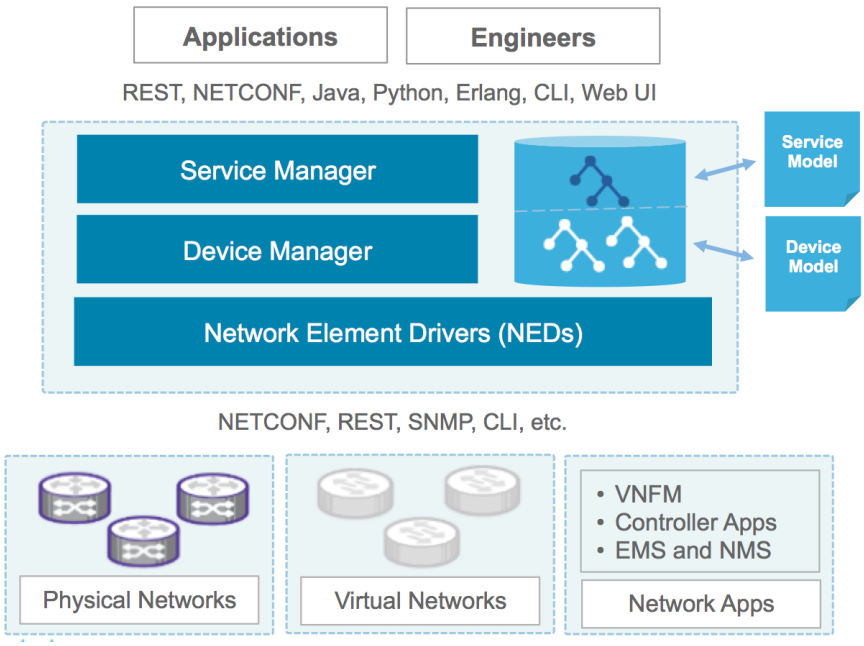
Time-to-market requirements are critical for a new service to be deployed quickly and the delay in configuring the corresponding tools has a significant impact on revenue. There is an unserved need in provider networks for tools which address these complex and sometimes contradictory challenges while constructing service configurations.

How NSO Addresses the Orchestration Challenge

NSO enables service providers to dynamically adopt the orchestration solution according to changes in the offered service portfolio. This is enabled by using a model-driven architecture where service definitions can be changed on the fly. Rather then a hard-coded orchestrator, NSO lears from the service models. Service models are written in YANG (RFC 6020).

NSO delivers an automated orchestration solution towards hybrid multi-vendor network. The network can be a mix of traditional equipment, virtual devices and SDN Controllers. This flexibility is managed by a Network Element Driver, NED, layer that abstracts the device interfaces and the Device Manager which enables generic device configuration functions.

Figure 3. NSO Logical Architecture



At the core of NSO is the configuration datastore, CDB, that is in sync with the actual device and service configuration. It also manages relationships between services and devices and can handle revisions of device interfaces.

The *Service Manager* addresses the following challenges:

- Transaction-safe activation of services across different multi-vendor devices.
- What-if scenarios, (dry-run), showing the effects on the network for a service creation/change.
- Maintaining relationships between services and corresponding device configurations and vice versa.
- Modeling of services
- Short development and turn-around time for new services.
- Mapping the service model to device models.

The *Device Manager* supports the following overall features:

- Deploy configuration changes to multiple devices in a fail-safe way using distributed transactions.
- Validate the integrity of configurations before deploying to the network.
- Apply configuration changes to named device groups.
- Apply templates (with variables) to named device groups.
- Easily roll back changes, if needed.
- Configuration audits: Check if device configurations are in synch with the NSO database. If they are not, what is the diff?
- Synchronize the NSO database and the configurations on devices, in case they are not in synch. This can be done in either direction (import the diff to the NSO database or deploy the diff on devices).

NSO provides user interfaces as well as northbound APIs for integration to other systems. The main user interface is the NSO network-wide CLI which gives a unified CLI towards the complete network including the network services. This User Guide will illustrate most of the functions using the CLI. NSO also provides a Web UI.

The northbound APIs are available in different language bindings (Java, Python), and as protocols, like NETCONF and REST.

In order to support dynamic updates of functionality as added or modified service models, support for a new device type etc, NSO manages extensions as well defined packages. Every NED is its own package with its own release life-cycle. Every service model with corresponding mapping is also a package of its own. These can be upgraded without upgrading NSO itself.

Note When running NSO against real devices, (not just the NSO network simulator for educational purposes), make sure you have the correct NED package version from the delivery repository.

In order to learn how to use NSO and also to simplify development using NSO, NSO comes with a network simulator, **ncs-netsim**. Many of the examples will use netsim as the network.

High-Availability and Clustering

NSO supports a 1:M high-availability mode. One NSO system can be primary and it can have any number of secondaries. Any configuration write has to go through the primary node. The configuration changes are replicated to the read-only secondaries. The replication can be done in asynchronous or synchronous mode. In the synchronous the transaction returns when the secondaries are in sync.

For large networks the network devices can be clustered across NSO systems. Say you have 100 000 devices split in two continents. You may choose to have 50 000 devices in one NSO and 50 000 in another. There are several options on how to configure clusters to see the whole network. The most common is a top NSO where also services are provisioned, and the top NSO sees the whole network.

CHAPTER 3

Installation

You can install NSO in production or development use. Understand the different install types to decide which deployment suits your needs.

Install Types

The installation of NSO comes in two variants, i.e.

- Local Install
- System Install

Before proceeding with the installation, decide the install type.

The following sections provide more information on the install types to help you choose between the two.

Local Install

Local install is used for development, lab, and evaluation purposes. It unpacks all the application components (including docs and examples) and can be used by the engineer to run multiple, unrelated instances of NSO on a single workstation for different labs and demos.

Note All the NSO examples and README steps provided with the installation are based on Local Install only. Use Local Install for evaluation and development purposes only.

System Install

System Install is used when installing NSO for a centralized, always-on, production-grade, system-wide deployment. It is configured as a system daemon that starts and ends with the underlying operating system. The default users of Admin and Operator are not included and the file structure is more distributed.

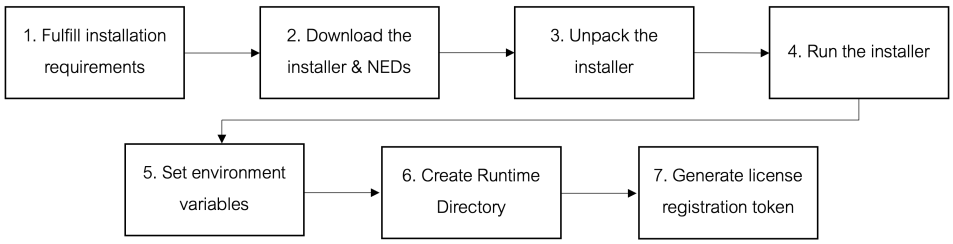
System Install should be used only for production deployment. For all other purposes, use Local Install.

Note All the NSO examples and README steps are primarily intended for Local Install and cannot run on a System Install out of the box. To have the examples run on a System Install, you need to make certain modifications first, as described in the section called “Modify Examples for System Install”.

Local Install Steps

The installation process involves a number of activities that must be completed in the order shown below.

Figure 4. Local Install Flow



An overview of steps is given below.

Table 5. Overview of Steps

Prepare	1. Fulfill Installation Requirements
	2. Download the Installer and NEDs
	3. Unpack the Installer
Install	4. Run the Installer
Finalize	5. Set Environment Variables
	6. Create Runtime Directory
	7. Generate License Registration Token

1. Fulfill Installation Requirements

Start by setting up your system to install and run NSO.

To install NSO:

Step 1 Fulfill at least the primary requirements.

Step 2 If you intend to build and run NSO examples, you also need to install additional applications listed under Additional Requirements.

Primary Requirements

Primary requirements to do a Local Install include:

- Linux for `x86_64` architecture is required for production. Linux or macOS Darwin for `x86_64` or macOS Darwin for `arm64` is required for development purposes.
- GNU libc 2.24 or higher.
- Java JRE 17 or higher. Used by Cisco Smart Licensing.
- Required and included with many Linux/macOS distributions:
  - **tar** command. Unpack the installer.
  - **gzip** command. Unpack the installer.
  - **ssh-keygen** command. Generate SSH hostkey.
  - **openssl** command. Generate self-signed certificates for HTTPS.
  - **find** command. Used to find out if all required libraries are available.
  - **which** command. Used by the NSO package manager.
  - libpam.so.0. Pluggable Authentication Module library.
  - libexpat.so.1. EXtensible Markup Language parsing library.
  - libz.so.1 version 1.2.7.1 or higher. Data compression library.

Additional Requirements

Additional requirements to, for example, build and run NSO examples, service, include:

- Java JDK 17 or higher.
- Ant 1.9.3 or higher.
- Python 3.7 or higher.
- Python Setuptools is required to build the Python API.
- Often installed using the Python package installer pip:
  - Python Paramiko 2.2 or higher. To use netconf-console.
  - Python requests. Used by the RESTCONF demo scripts.
- **xsltproc** command. Used by the support/ned-make-package-meta-data command to generate the package-meta-data.xml file.
- One of the following web browsers is required for NSO GUI capabilities. The version must be supported by the vendor at the time of release.
  - Safari
  - Mozilla Firefox
  - Microsoft Edge
  - Google Chrome
- OpenSSH client applications. For example **ssh** and **scp** commands.

2. Download the Installer and NEDs

To download the Cisco NSO installer and example NEDs:

Step 1 Go to the Cisco's official Software Download site.

Step 2 Search for product "Network Services Orchestrator" and select the desired version.

Step 3 There are two versions of the NSO installer, i.e. for macOS and Linux systems. Download the desired installer.

Identifying the Installer

You need to know your system specifications (Operating System and CPU architecture) in order to choose the appropriate NSO Installer.

NSO is delivered as an OS/CPU specific signed self-extractable archive. The signed archive file has the pattern `nso-VERSION.OS.ARCH.signed.bin` that after signature verification extracts the `nso-VERSION.OS.ARCH.installer.bin` archive file, where:

- `VERSION` is the NSO version to install.
- `OS` is the Operating System (`linux` for all Linux distributions and `darwin` for macOS).
- `ARCH` is the CPU architecture, for example, `x86_64`.

3. Unpack the Installer

If your downloaded file is a `signed.bin` file, it means that it has been digitally signed by Cisco, and upon execution, you will verify the signature and unpack the `installer.bin`.

If you only have `installer.bin`, skip to next step.

To unpack the installer:

Step 1 In the terminal, list the binaries in the directory where you downloaded the installer, for example:

```
cd ~/Downloads
ls -l nso*.bin
-rw-r--r--@ 1 user  staff   199M Dec 15 11:45 nso-6.0.darwin.x86_64.installer.bin
-rw-r--r--@ 1 user  staff   199M Dec 15 11:45 nso-6.0.darwin.x86_64.signed.bin
```

Step 2 Use the `sh` command to run the `signed.bin` to verify the certificate and extract the installer binary and other files. An example output is shown below.

```
sh nso-6.0.darwin.x86_64.signed.bin
# Output
Unpacking...
Verifying signature...
Downloading CA certificate from http://www.cisco.com/security/pki/certs/crcam2.cer ...
Successfully downloaded and verified crcam2.cer.
Downloading SubCA certificate from http://www.cisco.com/security/pki/certs/innerspace.cer ...
Successfully downloaded and verified innerspace.cer.
Successfully verified root, subca and end-entity certificate chain.
Successfully fetched a public key from tailf.cer.
Successfully verified the signature of nso-6.0.darwin.x86_64.installer.bin using tailf.cer
```

Step 3 List the files to check if extraction was successful.

```
ls -l
# Output
-rw-r--r--  1 user  staff   1.8K Nov 29 06:05 README.signature
-rw-r--r--  1 user  staff    12K Nov 29 06:05 cisco_x509_verify_release.py
-rwxr-xr-x  1 user  staff   199M Nov 29 05:55 nso-6.0.darwin.x86_64.installer.bin
-rw-r--r--  1 user  staff   256B Nov 29 06:05 nso-6.0.darwin.x86_64.installer.bin.signature
```

```
-rwxr-xr-x@ 1 user  staff   199M Dec 15 11:45 nso-6.0.darwin.x86_64.signed.bin
-rw-r--r--  1 user  staff   1.4K Nov 29 06:05 tailf.cer
```

Description of Unpacked Files

The following contents are unpacked:

- `nso-VERSION.OS.ARCH.installer.bin`: The NSO installer.
- `nso-VERSION.OS.ARCH.installer.bin.signature`: Signature generated for the NSO image.
- `tailf.cer`: An enclosed Cisco signed x.509 end-entity certificate containing the public key that is used to verify the signature.
- `README.signature`: File with further details on the unpacked content and steps on how to run the signature verification program. To manually verify the signature, refer to the steps in this file.
- `cisco_x509_verify_release.py`: Python program that can be used to verify the 3-tier x.509 certificate chain and signature.

NED Packages

The NED Packages that are available with the NSO Installation are netsim-based example NEDs. These NEDs are used for NSO examples only.

Fetch the latest production-grade NEDs from Cisco Software Download using the URLs provided on your NED license certificates.

Installer Help

Run the `sh nso-VERSION.darwin.x86_64.installer.bin --help` command to view additional help on running binaries. More details can be found in the ncs-installer(1) manual page included with NSO and also available from the online documentation on DevNet.

Notice the two options for `--local-install` or `--system-install`. An example output is shown below.

```
sh nso-6.0.darwin.x86_64.installer.bin --help

# Output
This is the NCS installation script.
Usage: ./nso-6.0.darwin.x86_64.installer.bin [--local-install] LocalInstallDir
Installs NCS in the LocalInstallDir directory only.
This is convenient for test and development purposes.
Usage: ./nso-6.0.darwin.x86_64.installer.bin --system-install
[--install-dir InstallDir]
[--config-dir ConfigDir] [--run-dir RunDir] [--log-dir LogDir]
[--run-as-user User] [--keep-ncs-setup] [--non-interactive]

Does a system install of NCS, suitable for deployment.
Static files are installed in InstallDir/ncs-<vsn>.
The first time --system-install is used, the ConfigDir,
RunDir, and LogDir directories are also created and
populated for config files, run-time state files, and log files,
respectively, and an init script for start of NCS at system boot
and user profile scripts are installed. Defaults are:

InstallDir - /opt/ncs
ConfigDir  - /etc/ncs
RunDir     - /var/opt/ncs
```

```
LogDir     - /var/log/ncs

By default, the system install will run NCS as the root user.
If the --run-as-user option is given, the system install will
instead run NCS as the given user. The user will be created if
it does not already exist.
If the --non-interactive option is given, the installer will
proceed with potentially disruptive changes (e.g. modifying or
removing existing files) without asking for confirmation.
```

4. Run the Installer

Local Install of NSO Software is performed in a single user-specified directory, for example in your `$HOME` directory. It is always recommended to install NSO in a directory named as the version of the release, for example, if the version being installed is `6.0`, the directory should be `~/nso-6.0`.

To run the installer:

Step 1 Navigate to your Install Directory.

Step 2 Run the following command to install NSO in your Install Directory. The `--local-install` parameter is optional.

```
$ sh nso-VERSION.OS.ARCH.installer.bin $HOME/ncs-VERSION --local-install
```

An example output is shown below.

```
sh nso-6.0.darwin.x86_64.installer.bin --local-install ~/nso-6.0

# Output
INFO  Using temporary directory /var/folders/90/n5sbctr922336_
0jrzhb54400000gn/T//ncs_installer.93831 to stage NCS installation bundle
INFO  Unpacked ncs-6.0 in /Users/user/nso-6.0
INFO  Found and unpacked corresponding DOCUMENTATION_PACKAGE
INFO  Found and unpacked corresponding EXAMPLE_PACKAGE
INFO  Found and unpacked corresponding JAVA_PACKAGE
INFO  Generating default SSH hostkey (this may take some time)
INFO  SSH hostkey generated
INFO  Environment set-up generated in /Users/user/nso-6.0/ncsrc
INFO  NSO installation script finished
INFO  Found and unpacked corresponding NETSIM_PACKAGE
INFO  NCS installation complete
```

Manual Pages

The installation program unpacks the NSO manual pages from the documentation archive in `$NCS_DIR/man`. `ncsrc` makes an addition to `$MANPATH`, allowing you to use the `man` command to view them. The manual pages are also available in the PDF format and from the online documentation located on NCS man-pages, Volume 1 in *Manual Pages* .

Following is a list of few of the installed manual pages:

- ncs(1): Command to start and control the NSO daemon.
- ncsc(1): NSO Yang compiler.
- ncs_cli(1): Frontend to the NSO CLI engine.
- ncs-netsim(1): Command to create and manipulate a simulated network.
- ncs-setup(1): Command to create an initial NSO setup.

- ncs.conf: NSO daemon configuration file format.

For example, to view the manual page describing NSO configuration file you should type:

```
$ man ncs.conf
```

Apart from the manual pages, extensive information about command line options can be obtained by running `ncs` and `ncsc` with the `--help` (abbreviated `-h`) flag.

```
$ ncs --help
```

```
$ ncsc --help
```

5. Set Environment Variables

The installation program creates a shell script file named `ncsrc` in each NSO installation, which sets the environment variables.

To set the environment variables:

Step 1 Source the `ncsrc` file to get the environment variables settings in your shell. You may want to add this sourcing command to your login sequence, such as `.bashrc`.

For `csh/tcsh` users, there is a `ncsrc.tcsh` file with `csh/tcsh` syntax. The example below assumes that you are using `bash`, other versions of `/bin/sh` may require that you use `.` instead of `source`.

```
$ source $HOME/ncs-VERSION/ncsrc
```

Step 2 Most users add source `~/nso-x.x/ncsrc` (where x.x is the NSO version) to their `~/.bash_profile`, but you can simply do it manually when you want it.

Once it has been sourced, you have access to all the NSO executable commands, which start with `ncs`.

```
ncs {TAB} {TAB}

# Output
ncs         ncs-maapi        ncs-project  ncs-start-python-vm  ncs_cmd
ncs-backup  ncs-make-package ncs-setup    ncs-uninstall        ncs_conf_tool
ncs-collect ncs-netsim       ncs-start-java-vm                 ncs_cli

ncs_load
ncsc
ncs_crypto_keys-tech-report
```

6. Create Runtime Directory

NSO needs a Deployment/Runtime Directory where the database files, logs, etc. are stored. An empty default directory can be created using the `ncs-setup` command.

To create a Runtime Directory:

Step 1 Create a Runtime Directory for NSO by running the following command. In this case, we assume that the directory is `$HOME/ncs-run`.

```
$ ncs-setup --dest $HOME/ncs-run
```

Step 2 Start the NSO daemon `ncs`.

```
$ cd $HOME/ncs-run
```

```
$ ncs
```

Runtime vs. Installation Directory

A common misunderstanding is that there exists a dependency between the Runtime Directory and the Installation Directory. This is not true. For example, say that you have two NSO installations `.../ncs-2.3.1` and `.../ncs-2.3.2`. The following sequence runs `ncs-2.3.1` but using an example and configuration from `ncs-2.3.2`.

```
$ cd .../ncs-2.3.1
$ . ncsrc
$ cd .../ncs-2.3.2/examples.ncs/datacenter-qinq
$ ncs
```

Since the Runtime Directory is self-contained, this is also the way to move between examples. And since the Runtime Directory is self-contained including the database files, you can compress a complete directory and distribute it. Unpacking that directory and starting NSO from there gives an exact copy of all instance data.

```
$ cd $NCS_DIR/examples.ncs/data-center-qinq
$ ncs
$ ncs --stop
$ cd $NCS_DIR/examples.ncs/getting-started/1-simulated-cisco-ios
$ ncs
$ ncs --stop
```

7. Generate License Registration Token

To conclude the NSO installation, a license registration token must be created using a (CSSM) account. This is because NSO uses Cisco Smart Licensing, as described in Chapter 3, *Cisco Smart Licensing* in *Administration Guide* to make it easy to deploy and manage NSO license entitlements. Login credentials to Cisco Smart Software Manager (CSSM) account are provided by your Cisco contact and detailed instructions on how to create a registration token can be found in Chapter 3, *Cisco Smart Licensing* in *Administration Guide*. General licensing information covering licensing models, how licensing works, usage compliance, etc., is covered in the Cisco Software Licensing Guide.

To generate a license registration token:

Step 1 When you have a token, start a Cisco CLI towards NSO and enter the token, for example:

```
admin@ncs# license smart register idtoken YzIzMDM3MTgtZTRkNC00YjkxLTk2ODQt
OGEzMTM3OTg5MG
Registration process in progress.
Use the 'show license status' command to check the progress and result.
```

Upon successful registration, NSO automatically requests a license entitlement for its own instance and for the number of devices it orchestrates and their NED types. If development mode has been enabled, only development entitlement for the NSO instance itself is requested.

Step 2 Inspect the requested entitlements using the command `show license all` (or by inspecting the NSO daemon log). An example output is shown below.

```
admin@ncs# show license all
...
<INFO> 21-Apr-2016::11:29:18.022 miosaterm confd[8226]:
Smart Licensing Global Notification:
type = "notifyRegisterSuccess",
```

```
agentID = "sa1",
enforceMode = "notApplicable",
allowRestricted = false,
failReasonCode = "success",
failMessage = "Successful."
<INFO> 21-Apr-2016::11:29:23.029 miosaterm confd[8226]:
Smart Licensing Entitlement Notification: type = "notifyEnforcementMode",
agentID = "sa1",
notificationTime = "Apr 21 11:29:20 2016",
version = "1.0",
displayName = "regid.2015-10.com.cisco.NSO-network-element",
requestedDate = "Apr 21 11:26:19 2016",
tag = "regid.2015-10.com.cisco.NSO-network-element",
enforceMode = "inCompliance",
daysLeft = 90,
expiryDate = "Jul 20 11:26:19 2016",
requestedCount = 8
...
```

Evaluation Period

If no registration token is provided, NSO enters a 90 days evaluation period and the remaining evaluation time is recorded hourly in the NSO daemon log:

```
...
<INFO> 13-Apr-2016::13:22:29.178 miosaterm confd[16260]:
    Starting the NCS Smart Licensing Java VM
<INFO> 13-Apr-2016::13:22:34.737 miosaterm confd[16260]:
Smart Licensing evaluation time remaining: 90d 0h 0m 0s
...
<INFO> 13-Apr-2016::13:22:34.737 miosaterm confd[16260]:
  Smart Licensing evaluation time remaining: 89d 23h 0m 0s
...
```

Communication Send Error

During upgrades, if you experience the 'Communication Send Error' with license registration, restart the Smart Agent.

Unable to Access Cisco Smart Software Manager?

In a situation where NSO instance has no direct access to the Cisco Smart Software Manager, one option is Cisco Smart Software Manager Satellite that can be installed to manage software licenses on the premises. Install the satellite and use the command `call-home destination address http <url:port>` to point to the satellite.

Another option when direct access is not desired is to configure an HTTP or HTTPS proxy, e.g., `smart-license smart-agent proxy url https://127.0.0.1:8080`. If you plan to do this, take the note below regarding ignored CLI configurations into account:

If `ncs.conf` contains configuration for any of java-executable, java-options, override-url/url or proxy/url under the configure path `/ncs-config/smart-license/smart-agent/`, then any corresponding configuration done via the CLI is ignored.

License Registration in High Availability (HA) Mode

When configuring NSO in HA mode, the license registration token must be provided to the CLI running on the primary node. Read more about HA and node types in NSO Chapter 7, *High Availability* in *Administration Guide*.

Licensing Log

Licensing activities are also logged in the NSO daemon log as described in the section called “Monitoring NSO” in *Administration Guide*. For example, a successful token registration results in the following log entry:

```
<INFO> 21-Apr-2016::11:29:18.022 miosaterm confd[8226]:
  Smart Licensing Global Notification:
  type = "notifyRegisterSuccess"
```

Check Registration Status

To check registration status, use the command `show license status` An example output is shown below..

```
admin@ncs# show license status
Smart Licensing is ENABLED

Registration:
Status: REGISTERED
Smart Account: Network Service Orchestrator
Virtual Account: Default
Export-Controlled Functionality: Allowed
Initial Registration: SUCCEEDED on Apr 21 09:29:11 2016 UTC
Last Renewal Attempt: SUCCEEDED on Apr 21 09:29:16 2016 UTC
Next Renewal Attempt: Oct 18 09:29:16 2016 UTC
Registration Expires: Apr 21 09:26:13 2017 UTC
Export-Controlled Functionality: Allowed

License Authorization:
License Authorization:
Status: IN COMPLIANCE on Apr 21 09:29:18 2016 UTC
Last Communication Attempt: SUCCEEDED on Apr 21 09:26:30 2016 UTC
Next Communication Attempt: Apr 21 21:29:32 2016 UTC
Communication Deadline: Apr 21 09:26:13 2017 UTC
```

Explore the Installation

Before staring NSO, it is recommended to explore the installation contents.

Navigate to the newly created Installation Directory, for example:

```
cd ~/nso-6.0
```

The Installation Directory includes the following contents:

- Documentation
- Examples
- Network Element Drivers
- Shell scripts

Documentation

Along with the binaries, NSO installs a full set of documentation available in the in the `doc/` folder in the Installation Directory. There is also an online version of the documentation available on DevNet.

```
ls -l doc/
drwxr-xr-x   5 user  staff   160B Nov 29 05:19 api/
drwxr-xr-x  14 user  staff   448B Nov 29 05:19 html/
```

```
-rw-r--r--   1 user  staff   202B Nov 29 05:19 index.html
drwxr-xr-x  17 user  staff   544B Nov 29 05:19 pdf/
```

Run `index.html` in your browser to explore further.

Examples

Local Install comes with a rich set of examples to start using NSO.

```
$ ls -1 examples.ncs/
README
crypto
datacenter
development-guide
generic-ned
getting-started
misc
service-provider
snmp-ned
snmp-notification-receiver
web-server-farm
web-ui
```

Network Element Drivers

In order to communicate with the network, NSO uses NEDs as device drivers for different device types. Cisco has NEDs for hundreds of different devices available for customers, and several are included in the installer in the `/packages/neds` directory.

In the example below, NEDs for Cisco ASA, IOS, IOS XR, and NX-OS are shown. Also included are NEDs for other vendors including Juniper JunOS, A10, ALU, and Dell.

```
$ ls -1 packages/neds
a10-acos-cli-3.0
alu-sr-cli-3.4
cisco-asa-cli-6.6
cisco-ios-cli-3.0
cisco-ios-cli-3.8
cisco-iosxr-cli-3.0
cisco-iosxr-cli-3.5
cisco-nx-cli-3.0
dell-ftos-cli-3.0
juniper-junos-nc-3.0
```

Note The example NEDs included in the installer are intended for evaluation, demonstration, and use with the `examples.ncs`. These are not the latest versions available, and often do not have all the features available in production NEDs.

Install new NEDs

A large number of pre-built supported NEDs are available which can be acquired and downloaded by the customers from Cisco Software Download. Note that the specific file names and versions that you download may be different from the ones in this guide. Therefore, remember to update the paths accordingly.

Like the NSO installer, the NEDs are `signed.bin` files that need to be run to validate the download and extract the new code.

To install new NEDs:

Step 1 Change to the working directory where your downloads are. The filenames indicate which version of NSO the NEDs are pre-compiled for (in this case NSO 6.0), and the version of the NED. An example output is shown below.

```
cd ~/Downloads/
ls -l ncs*.bin

# Output
-rw-r--r--@ 1 user  staff   9708091 Dec 18 12:05 ncs-6.0-cisco-asa-6.7.7.signed.bin
-rw-r--r--@ 1 user  staff  51233042 Dec 18 12:06 ncs-6.0-cisco-ios-6.42.1.signed.bin
-rw-r--r--@ 1 user  staff   8400190 Dec 18 12:05 ncs-6.0-cisco-nx-5.13.1.1.signed.bin
```

Step 2 Use the `sh` command to run `signed.bin` to verify the certificate and extract the NED tar.gz and other files. Repeat for all files. An example output is shown below.

```
sh ncs-6.0-cisco-nx-5.13.1.1.signed.bin

  Unpacking...
  Verifying signature...
  Downloading CA certificate from http://www.cisco.com/security/pki/certs/crcam2.cer ...
  Successfully downloaded and verified crcam2.cer.
  Downloading SubCA certificate from http://www.cisco.com/security/pki/certs/innerspace.cer ...
  Successfully downloaded and verified innerspace.cer.
  Successfully verified root, subca and end-entity certificate chain.
  Successfully fetched a public key from tailf.cer.
  Successfully verified the signature of ncs-6.0-cisco-nx-5.13.1.1.tar.gz using tailf.cer
```

Step 3 You now have 3 tar (.tar.gz) files. These are compressed versions of the NEDs. List the files to verify as shown in the example below.

```
ls -l ncs*.tar.gz
-rw-r--r--  1 user  staff   9704896 Dec 12 21:11 ncs-6.0-cisco-asa-6.7.7.tar.gz
-rw-r--r--  1 user  staff  51260488 Dec 13 22:58 ncs-6.0-cisco-ios-6.42.1.tar.gz
-rw-r--r--  1 user  staff   8409288 Dec 18 09:09 ncs-6.0-cisco-nx-5.13.1.1.tar.gz
```

Step 4 Navigate to the `packages/neds` directory for your Local Install, for example:

```
cd ~/nso-6.0/packages/neds
```

Step 5 In the `/packages/neds` directory, extract the .tar files into this directory using the `tar` command with the path to where the compressed NED is located. An example is shown below.

```
tar -zxvf ~/Downloads/ncs-6.0-cisco-nx-5.13.1.1.tar.gz
tar -zxvf ~/Downloads/ncs-6.0-cisco-ios-6.42.1.tar.gz
tar -zxvf ~/Downloads/ncs-6.0-cisco-asa-6.7.7.tar.gz
```

Here is a sample list of the newer NEDs extracted along with the ones bundled with the installation:

```
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 a10-acos-cli-3.0
drwxr-xr-x  12 user  staff   384 Nov 29 05:17 alu-sr-cli-3.4
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 cisco-asa-cli-6.6
drwxr-xr-x  13 user  staff   416 Dec 12 21:11 cisco-asa-cli-6.7
drwxr-xr-x  12 user  staff   384 Nov 29 05:17 cisco-ios-cli-3.0
drwxr-xr-x  12 user  staff   384 Nov 29 05:17 cisco-ios-cli-3.8
drwxr-xr-x  13 user  staff   416 Dec 13 22:58 cisco-ios-cli-6.42
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 cisco-iosxr-cli-3.0
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 cisco-iosxr-cli-3.5
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 cisco-nx-cli-3.0
drwxr-xr-x  14 user  staff   448 Dec 18 09:09 cisco-nx-cli-5.13
drwxr-xr-x  13 user  staff   416 Nov 29 05:17 dell-ftos-cli-3.0
drwxr-xr-x  10 user  staff   320 Nov 29 05:17 juniper-junos-nc-3.0
```

Shell Scripts

The last thing to note are the files `ncsrc` and `ncsrc.tsch`. These are shell scripts for bash and tsch that setup your PATH and other environment variables for NSO. Depending on your shell, you need to source this file before starting NSO.

For more information on sourcing shell script, see the Local Install steps.

Start and Stop NSO

The command `ncs -h` shows various options when starting NSO. By default, NSO starts in the background without an associated terminal. It is recommended to add NSO to the `/etc/init` scripts of the deployment hosts. For more information, see the ncs(1) in *Manual Pages* .

Whenever you start (or reload) the NSO daemon, it reads its configuration from `./ncs.conf` or `${NCS_DIR}/etc/ncs/ncs.conf` or from the file specified with the `-c` option.

```
$ ncs
$ ncs --stop
$ ncs -h
...
```

Enable Development Mode

If you intend to use your NSO instance for development purposes, enable the development mode using the command `license smart development enable`.

Create NSO Instance

One of the included scripts with an NSO installation is `ncs-setup`, which makes it very easy to create instances of NSO from a Local Install. You can look at the `--help` or ncs-setup(1) in *Manual Pages* for more details, but the two options we need to know are:

- `--dest` defines the directory where you want to set up NSO. if the directory does not exist, it will be created.
- `--package` defines the NEDs that you want to have installed. You can specify this option multiple times.

Note NCS is the original name of the NSO product. Therefore, many of the commands and application features are prefaced with `ncs`. You can think of NCS as another name for NSO.

To create an NSO instance:

Step 1 Run the command to set up an NSO instance in the current directory with the IOS, NX-OS, IOS-XR and ASA NEDs. You only need one NED per platform that you want NSO to manage, even if you may have multiple versions in your installer `neds` directory.

Use the name of the NED folder in `${NCS_DIR}/packages/neds` for the latest NED version that you have installed for the target platform. Use the tab key to complete the path, after you start typing (alternatively, copy and paste). Verify that the NED versions below match what is currently on the sandbox to avoid a syntax error. See example below.

```
ncs-setup --package ~/nso-6.0/packages/neds/cisco-ios-cli-6.44 \
```

```
--package ~/nso-6.0/packages/neds/cisco-nx-cli-5.15 \
--package ~/nso-6.0/packages/neds/cisco-iosxr-cli-7.20 \
--package ~/nso-6.0/packages/neds/cisco-asa-cli-6.8 \
--dest nso-instance
```

Step 2 Check the `nso-instance` directory. Notice that several new files and folders are created.

```
$ ls nso-instance/
logs  ncs-cdb  ncs.conf  packages  README.ncs  scripts  state
$ ls -l nso-instance/packages/
total 0
lrwxrwxrwx 1 user docker 51 Mar 19 12:44 cisco-asa-cli-6.8 ->
/home/user/nso-6.0/packages/neds/cisco-asa-cli-6.8

lrwxrwxrwx 1 user docker 52 Mar 19 12:44 cisco-ios-cli-6.44 ->
/home/user/nso-6.0/packages/neds/cisco-ios-cli-6.44

lrwxrwxrwx 1 user docker 54 Mar 19 12:44 cisco-iosxr-cli-7.20 ->
/home/user/nso-6.0/packages/neds/cisco-iosxr-cli-7.20

lrwxrwxrwx 1 user docker 51 Mar 19 12:44 cisco-nx-cli-5.15 ->
/home/user/nso-6.0/packages/neds/cisco-nx-cli-5.15
$
```

Following is a description of the important files and folders:

- `ncs.conf` is the NSO application configuration file, and is used to customize aspects of the NSO instance (for example, to change ports, enable/disable features, and so on.) See ncs.conf(5) in *Manual Pages* for information.
- `packages/` is the directory that has symlinks to the NEDs that we referenced in the `--package` arguments at time of setup. See Chapter 17, *NSO Packages* in *Development Guide* for more information.
- `logs/` is the directory that contains all the logs from NSO. This directory is useful for troubleshooting.

Step 3 Start the NSO instance by navigating to the `nso-instance` directory and typing the `ncs` command .

You must be situated in the `nso-instance` directory each time you want to start or stop NSO. If you have multiple instances, you need to navigate to each one and use the `ncs` command to start or stop each one.

Step 4 Verify that NSO is running by using the `ncs --status | grep status` command.

```
$ ncs --status | grep status
status: started
db=running id=31 priority=1 path=/ncs:devices/device/live-status-protocol/device-type
```

Step 5 Add Netsim or lab devices using the command `ncs-netsim -h`.

Migrate to System Install

If you already have a Local Install with existing data that you would like to convert into a System Install, the following procedure allows you to do so. However, a reverse migration from System to Local Install is not supported.

Note It is possible to perform the migration and upgrade simultaneously to a newer NSO version, however, doing so introduces additional complexity. If you run into issues, first do the migration, and then perform the upgrade.

The following procedure assumes that NSO is installed as described in the NSO Local Install process, and will perform an initial System Install of the same NSO version. After following these steps, consult the NSO System Install guide for additional steps that are required for a fully functional System Install.

The procedure also assumes you are using the `$HOME/ncs-run` folder as the Run Directory. If this is not the case, modify the following path accordingly.

To migrate to System Install:

Step 1 Stop the current (local) NSO instance, if it is running.

```
$ ncs --stop
```

Step 2 Take a complete backup of the Runtime Directory for potential disaster recovery.

```
$ tar -czf $HOME/ncs-backup.tar.gz -C $HOME ncs-run
```

Step 3 Change to Super User privileges.

```
$ sudo -s
```

Step 4 Start the NSO System Install.

```
$ sh nso-VERSION.OS.ARCH.installer.bin --system-install
```

Step 5 If you have multiple versions of NSO installed, verify that the symbolic link in `/opt/ncs` points to the correct version.

Step 6 Copy the CDB files containing data to the central location.

```
# cp $HOME/ncs-run/ncs-cdb/*.cdb /var/opt/ncs/cdb
```

Step 7 Ensure that the `/var/opt/ncs/packages` directory includes all the necessary packages, appropriate for the NSO version. However, copying the packages directly could later on interfere with the operation of the `nct` command. It is better to only use symbolic links in that folder. Instead, copy the existing packages to the `/opt/ncs/packages` directory, either as directories or as tarball files. Make sure that each package includes the NSO version in its name and is not just a symlink, for example:

```
# cd $HOME/ncs-run/packages
# for pkg in *; do cp -RL $pkg /opt/ncs/packages/ncs-VERSION-$pkg; done
```

Step 8 Link to these packages in the `/var/opt/ncs/packages` directory.

```
# cd /var/opt/ncs/packages/
# rm -f *
# for pkg in /opt/ncs/packages/ncs-VERSION-*; do ln -s $pkg; done
```

The reason for prepending `ncs-VERSION` to the filename is to allow additional NSO commands, such as `nct upgrade` and `software packages` to work properly. These commands need to identify which NSO version a package was compiled for.

Step 9 Edit the `/etc/ncs/ncs.conf` configuration file and make the necessary changes. If you wish to use the configuration from Local Install, disable the local authentication, unless you fully understand its security implications.

```
<local-authentication>
  <enabled>false</enabled>
</local-authentication>
```

Step 10 When starting NSO later on, make sure that you set the package reload option, or use `start-with-package-reload` instead of `start` with `/etc/init.d/ncs`.

```
# export NCS_RELOAD_PACKAGES=true
```

Step 11 Review and complete the steps in NSO System Install, except running the installer, which you have done already. Once completed, you should have a running NSO instance with data from the Local Install.

Step 12 Remove the package reload option if it was set.

```
# unset NCS_RELOAD_PACKAGES
```

Step 13 Update log file paths for Java and Python VM through the NSO CLI.

```
$ ncs_cli -C -u admin
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# unhide debug
admin@ncs(config)# show full-configuration java-vm stdout-capture file
java-vm stdout-capture file ./logs/ncs-java-vm.log
admin@ncs(config)# java-vm stdout-capture file /var/log/ncs/ncs-java-vm.log
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)# show full-configuration java-vm stdout-capture file
java-vm stdout-capture file /var/log/ncs/ncs-java-vm.log
admin@ncs(config)# show full-configuration python-vm logging log-file-prefix
python-vm logging log-file-prefix ./logs/ncs-python-vm
admin@ncs(config)# python-vm logging log-file-prefix /var/log/ncs/ncs-python-vm
admin@ncs(config)# commit
Commit complete.
admin@ncs(config)# show full-configuration python-vm logging log-file-prefix
python-vm logging log-file-prefix /var/log/ncs/ncs-python-vm
admin@ncs(config)# exit
admin@ncs#
admin@ncs# exit
```

Step 14 Verify that everything is working correctly.

At this point you should have a complete copy of the previous Local Install running as a System Install. Should the migration fail at some point and you want to back out of it, the Local Install was not changed and you can easily go back to using it as before.

```
$ sudo /etc/init.d/ncs stop
$ source $HOME/ncs-VERSION/ncsrc
$ cd $HOME/ncs-run
$ ncs
```

In the unlikely event of Local Install becoming corrupted, you can restore it from the backup.

```
$ rm -rf $HOME/ncs-run
$ tar -xzf $HOME/ncs-backup.tar.gz -C $HOME
```

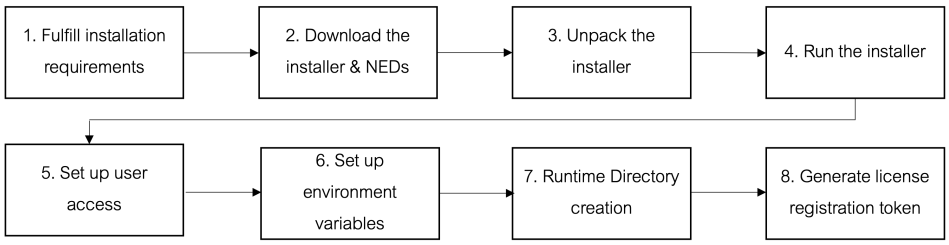
Uninstall NSO Local Install

To uninstall Local Install, delete the Install Directory.

System Install Steps

The installation process involves a number of activities that must be completed in the order shown below.

Figure 6. System Install Flow



An overview of steps is given below.

Table 7. Overview of Steps

Prepare	1. Fulfill Installation Requirements 2. Download the Installer and NEDs 3. Unpack the Installer
Install	4. Run the Installer
Finalize	5. Set up User Access 6. Set Environment Variables 7. Runtime Directory creation 8. Generate License Registration Token

1. Fulfill Installation Requirements

Start by setting up your system to install and run NSO.

To install NSO:

Step 1 Fulfill at least the primary requirements.

Step 2 If you intend to build and run NSO deployment examples, you also need to install additional applications listed under Additional Requirements.

Primary Requirements

Primary requirements to do a System Install include:

- Linux for `x86_64` architecture is required for production. Linux or macOS Darwin for `x86_64` or macOS Darwin for `arm64` is required for development purposes.
- GNU libc 2.24 or higher.
- Java JRE 17 or higher. Used by Cisco Smart Licensing.
- Required and included with many Linux/macOS distributions:
  - **tar** command. Unpack the installer.
  - **gzip** command. Unpack the installer.
  - **ssh-keygen** command. Generate SSH hostkey.
  - **openssl** command. Generate self-signed certificates for HTTPS.
  - **find** command. Used to find out if all required libraries are available.
  - **which** command. Used by the NSO package manager.
  - libpam.so.0. Pluggable Authentication Module library.
  - libexpat.so.1. EXtensible Markup Language parsing library.
  - libz.so.1 version 1.2.7.1 or higher. Data compression library.

Additional Requirements

Additional requirements to, for example, build and run NSO production deployment examples, include:

- Java JDK 17 or higher.
- Ant 1.9.3 or higher.
- Python 3.7 or higher.
- Python Setuptools is required to build the Python API.
- Often installed using the Python package installer pip:
  - Python Paramiko 2.2 or higher. To use netconf-console.
  - Python requests. Used by the RESTCONF demo scripts.
- **xsltproc** command. Used by the support/ned-make-package-meta-data command to generate the package-meta-data.xml file.
- One of the following web browsers is required for NSO GUI capabilities. The version must be supported by the vendor at the time of release.
  - Safari
  - Mozilla Firefox
  - Microsoft Edge
  - Google Chrome
- OpenSSH client applications. For example **ssh** and **scp** commands.
- cron. Run time-based tasks, such as **logrotate**.
- **logrotate**. rotate, compress, and mail NSO and system logs.
- rsyslog. pass NSO logs to a local syslog managed by **rsyslogd** and pass logs to a remote node.
- systemd or init.d scripts to start and stop NSO.

2. Download the Installer and NEDs

To download the Cisco NSO installer and example NEDs:

Step 1 Go to the Cisco's official Software Download site.

Step 2 Search for product "Network Services Orchestrator" and select the desired version.

Step 3 There are two versions of the NSO installer, i.e. for macOS and Linux systems. For System Install, choose the Linux OS version.

Identifying the Installer

You need to know your system specifications (Operating System and CPU architecture) in order to choose the appropriate NSO Installer.

NSO is delivered as an OS/CPU specific signed self-extractable archive. The signed archive file has the pattern `nso-VERSION.OS.ARCH.signed.bin` that after signature verification extracts the `nso-VERSION.OS.ARCH.installer.bin` archive file, where:

- `VERSION` is the NSO version to install.
- `OS` is the Operating System (`linux` for all Linux distributions and `darwin` for macOS).
- `ARCH` is the CPU architecture, for example, `x86_64`.

3. Unpack the Installer

If your downloaded file is a `signed.bin` file, it means that it has been digitally signed by Cisco, and upon execution, you will verify the signature and unpack the `installer.bin`.

If you only have `installer.bin`, skip to next step.

To unpack the installer:

Step 1 In the terminal, list the binaries in the directory where you downloaded the installer, for example:

```
cd ~/Downloads
ls -l nso*.bin
-rw-r--r--@ 1 user  staff   199M Dec 15 11:45 nso-6.0.linux.x86_64.installer.bin
-rw-r--r--@ 1 user  staff   199M Dec 15 11:45 nso-6.0.linux.x86_64.signed.bin
```

Step 2 Use the `sh` command to run the `signed.bin` to verify the certificate and extract the installer binary and other files. An example output is shown below.

```
sh nso-6.0.linux.x86_64.signed.bin
# Output
Unpacking...
Verifying signature...
Downloading CA certificate from http://www.cisco.com/security/pki/certs/crcam2.cer ...
Successfully downloaded and verified crcam2.cer.
Downloading SubCA certificate from http://www.cisco.com/security/pki/certs/innerspace.cer ...
Successfully downloaded and verified innerspace.cer.
Successfully verified root, subca and end-entity certificate chain.
Successfully fetched a public key from tailf.cer.
Successfully verified the signature of nso-6.0.linux.x86_64.installer.bin using tailf.cer
```

Step 3 List the files to check if extraction was successful.

```
ls -l
# Output
-rw-r--r--  1 user  staff   1.8K Nov 29 06:05 README.signature
-rw-r--r--  1 user  staff    12K Nov 29 06:05 cisco_x509_verify_release.py
-rwxr-xr-x  1 user  staff   199M Nov 29 05:55 nso-6.0.linux.x86_64.installer.bin
-rw-r--r--  1 user  staff   256B Nov 29 06:05 nso-6.0.linux.x86_64.installer.bin.signature
-rwxr-xr-x@ 1 user  staff   199M Dec 15 11:45 nso-6.0.linux.x86_64.signed.bin
-rw-r--r--  1 user  staff   1.4K Nov 29 06:05 tailf.cer
```

Description of Unpacked Files

The following contents are unpacked:

- `nso-VERSION.OS.ARCH.installer.bin`: The NSO installer.
- `nso-VERSION.OS.ARCH.installer.bin.signature`: Signature generated for the NSO image.
- `tailf.cer`: An enclosed Cisco signed x.509 end-entity certificate containing the public key that is used to verify the signature.
- `README.signature`: File with further details on the unpacked content and steps on how to run the signature verification program. To manually verify the signature, refer to the steps in this file.
- `cisco_x509_verify_release.py`: Python program that can be used to verify the 3-tier x.509 certificate chain and signature.

NED Packages

The NED Packages that are available with the NSO Installation are netsim-based example NEDs. These NEDs are used for NSO examples only.

Fetch the latest production-grade NEDs from Cisco Software Download using the URLs provided on your NED license certificates.

Installer Help

Run the `sh nso-VERSION.linux.x86_64.installer.bin --help` command to view additional help on running binaries. More details can be found in the ncs-installer(1) manual page included with NSO and also available from the online documentation on DevNet.

Notice the two options for `--local-install` or `--system-install`.

```
sh nso-6.0.linux.x86_64.installer.bin --help
```

4. Run the Installer

To run the installer:

Step 1 Navigate to your Install Directory.

Step 2 Run the installer with `--system-install` option to perform System Install. This option creates an Install of NSO that is suitable for deployment.

```
$ sudo sh nso-VERSION.OS.ARCH.installer.bin --system-install
```

For example:

```
$ sudo sh nso-6.0.linux.x86_64.installer.bin --system-install
```

Default Directories

The System Install by default creates the following directories:

- The Installation Directory is created in `/opt/ncs`, where the distribution is available.
- The Configuration Directory is created in `/etc/ncs`, where the `ncs.conf` file, SSH keys, WebUI certificates are created.
- The Running Directory is created in `/var/opt/ncs`, where runtime state files, CDB database, and packages are created.
- The Log Directory is created in `/var/log/ncs`, where the log files are populated. Also, init scripts are created in `/etc/init.d/ncs` and system-wide environment variables are created in `/etc/profile.d/ncs.sh`.

For the `--system-install` option, you can also choose user-defined (non-default) Installation Directory, Configuration Directory, Running Directory, and Log Directory with `--install-dir`, `--config-dir`, `--run-dir` and `--log-dir` parameters, and specify that NSO should run as a different user than root with the `--run-as-user` parameter.

If you choose a non-default Installation Directory by using `--install-dir`, you need to specify `--install-dir` for subsequent installs and also for backup and restore.

For more information on `ncs-installer`, see the ncs-installer(1) man page.

For an extensive guide to NSO deployment, refer to Chapter 11, *Deployment Example* in *Administration Guide*.

Manual Pages

The installation program will unpack the NSO manual pages from the documentation archive, allowing you to use the `man` command to view them. The manual pages are also available in the PDF format and from the online documentation located on NCS man-pages, Volume 1 in *Manual Pages*

Following is a list of few of the installed manual pages:

- ncs(1): Command to start and control the NSO daemon.
- ncsc(1): NSO Yang compiler.
- ncs_cli(1): Frontend to the NSO CLI engine.
- ncs-netsim(1): Command to create and manipulate a simulated network.
- ncs-setup(1): Command to create an initial NSO setup.
- ncs.conf: NSO daemon configuration file format.

For example, to view the manual page describing NSO configuration file you should type:

```
$ man ncs.conf
```

Apart from the manual pages, extensive information about command line options can be obtained by running `ncs` and `ncsc` with the `--help` (abbreviated `-h`) flag.

```
$ ncs --help
```

```
$ ncsc --help
```

Disable Memory Overcommit

By default, the Linux kernel allows overcommit of memory. However, memory overcommit produces an unexpected and unreliable environment for NSO since the Linux Out Of Memory Killer, or OOM-killer, may terminate NSO without restarting it if the system is critically low on memory. Also, when the OOM-killer terminates NSO, no system dump file will be produced, and the debug information will be lost. Thus, it is *strongly recommended* that overcommit is disabled.

To achieve this with immediate effect, give the command:

```
# echo 2 > /proc/sys/vm/overcommit_memory
```

When `overcommit_memory = 2`, the `/proc/sys/vm/overcommit_ratio` parameter defines the percent of the physical RAM + swap space used. The default is "50", or 50%. This setting will underutilize RAM usage if the system has more physical RAM than 50%.

Setting the `overcommit_ratio` parameter to "100" will include any swap if present. On-disk memory (swap) gives the advantage of having more memory available in case an application needs more RAM than physically available momentarily. But it is usually slow, and thus best practice is to refrain from using the swap for NSO. To allocate physical RAM only, set the `overcommit_ratio` parameter to 100 * ((RAM - swap space) / RAM).

If the system's physical RAM (MemTotal) is less than or equal to the swap space (SwapTotal), using the swap cannot be avoided and the `overcommit_ratio` should be set to '100'.

Example 1 - physical RAM (MemTotal) > swap space (SwapTotal):

```
# cat /proc/meminfo | grep "MemTotal\|SwapTotal"
MemTotal:    8039352 kB
SwapTotal:   1048572 kB
```

Calculate the overcommit ratio:

```
100 * ((8039352-1048572)/8039352) = ~86.9%
```

To set both overcommit parameters with immediate effect:

```
# echo 2 > /proc/sys/vm/overcommit_memory
# echo 86.9 > /proc/sys/vm/overcommit_ratio
```

Example 2 - physical RAM (MemTotal) == swap space (SwapTotal):

```
# cat /proc/meminfo | grep "MemTotal\|SwapTotal"
MemTotal:    16000000 kB
SwapTotal:   16000000 kB
```

To set both overcommit parameters with immediate effect:

```
# echo 2 > /proc/sys/vm/overcommit_memory
# echo 100 > /proc/sys/vm/overcommit_ratio
```

To ensure the overcommit remains disabled after reboot, adjust the `overcommit_ratio` parameter to match your system memory and add the two lines to the `/etc/sysctl.conf` file. See the Linux sysctl.conf(5) manual page for details.

Refer to the Linux proc(5) manual page for more details on the `overcommit_memory` and `overcommit_ratio` parameters.

Note If NSO aborts due to failure to allocate memory, NSO will produce a system dump by default before aborting. When starting NSO from a non-root user, set the `NCS_DUMP` environment variable to point to a filename in a directory that the non-root user can access. The default setting is `NCS_DUMP=ncs_crash.dump`, where the file is written to the NSO run-time directory, typically `NCS_RUN_DIR=/var/opt/ncs`. If the user running NSO cannot write to the directory the `NCS_DUMP` environment variable points to, generating the system dump file will fail, and the debug information will be lost.

5. Set up User Access

The installation is configured for PAM authentication, with group assignment based on the OS group database (e.g. `/etc/group` file). Users that need access to NSO must belong to either the `ncsadmin` group (for unlimited access rights) or the `ncsoper` group (for minimal access rights).

To set up user access:

Step 1 To create the `ncsadmin` group, use the OS shell command:

```
# groupadd ncsadmin
```

Step 2 To create the `ncsoper` group, use the OS shell command:

```
# groupadd ncsoper
```

Step 3 To add an existing user to one of these groups, use the OS shell command:

```
# usermod -a -G 'groupname' 'username'
```

6. Set Environment Variables

To set environment variables:

Step 1 Change to Super User privileges.

```
$ sudo -s
```

Step 2 The installation program creates a shell script file in each NSO installation which sets environment variables needed to run NSO. With the `--system-install` option, by default, these settings are set on the shell. To explicitly set the variables, source `ncs.sh` or `ncs.csh` depending on your shell type.

```
# source /etc/profile.d/ncs.sh
```

Step 3 Start NSO.

```
# /etc/init.d/ncs start
```

Once you log on with the user that belongs to `ncsadmin` or `ncsoper`, you can directly access the CLI as shown below:

```
$ ncs_cli -C
```

7. Runtime Directory Creation

As part of the System Install, the NSO daemon `ncs` is automatically started at boot time. You do not need to create a Runtime Directory for System Install.

8. Generate License Registration Token

To conclude the NSO installation, a license registration token must be created using a (CSSM) account. This is because NSO uses Cisco Smart Licensing, as described in Chapter 3, *Cisco Smart Licensing* in *Administration Guide* to make it easy to deploy and manage NSO license entitlements. Login credentials to Cisco Smart Software Manager (CSSM) account are provided by your Cisco contact and detailed instructions on how to create a registration token can be found in Chapter 3, *Cisco Smart Licensing* in *Administration Guide*. General licensing information covering licensing models, how licensing works, usage compliance, etc., is covered in the Cisco Software Licensing Guide.

To generate license registration token:

Step 1 When you have a token, start a Cisco CLI towards NSO and enter the token, for example:

```
admin@ncs# license smart register idtoken
YzIzMDM3MTgtZTRkNC00YjkxLTk2ODQtOGEzMTM3OTg5MG

Registration process in progress.
Use the 'show license status' command to check the progress and result.
```

Upon successful registration, NSO automatically requests a license entitlement for its own instance and for the number of devices it orchestrates and their NED types. If development mode has been enabled, only development entitlement for the NSO instance itself is requested.

Step 2 Inspect the requested entitlements using the command `show license all` (or by inspecting the NSO daemon log). An example output is shown below.

```
admin@ncs# show license all
...
<INFO> 21-Apr-2016::11:29:18.022 miosaterm confd[8226]:
Smart Licensing Global Notification:
type = "notifyRegisterSuccess",
agentID = "sa1",
enforceMode = "notApplicable",
allowRestricted = false,
failReasonCode = "success",
failMessage = "Successful."
<INFO> 21-Apr-2016::11:29:23.029 miosaterm confd[8226]:
Smart Licensing Entitlement Notification: type = "notifyEnforcementMode",
agentID = "sa1",
notificationTime = "Apr 21 11:29:20 2016",
version = "1.0",
```

```
displayName = "regid.2015-10.com.cisco.NSO-network-element",
requestedDate = "Apr 21 11:26:19 2016",
tag = "regid.2015-10.com.cisco.NSO-network-element",
enforceMode = "inCompliance",
daysLeft = 90,
expiryDate = "Jul 20 11:26:19 2016",
requestedCount = 8
...
```

Evaluation Period

If no registration token is provided, NSO enters a 90 days evaluation period and the remaining evaluation time is recorded hourly in the NSO daemon log:

```
      ...
<INFO> 13-Apr-2016::13:22:29.178 miosaterm confd[16260]:
Starting the NCS Smart Licensing Java VM
<INFO> 13-Apr-2016::13:22:34.737 miosaterm confd[16260]:
Smart Licensing evaluation time remaining: 90d 0h 0m 0s
...
<INFO> 13-Apr-2016::13:22:34.737 miosaterm confd[16260]:
Smart Licensing evaluation time remaining: 89d 23h 0m 0s
...
```

Communication Send Error

During upgrades, If you experience 'Communication Send Error' during license registration, restart the Smart Agent.

Unable to access Cisco Smart Software Manager?

In a situation where NSO instance has no direct access to the Cisco Smart Software Manager, one option is Cisco Smart Software Manager Satellite that can be installed to manage software licenses on the premises. Install the satellite and use the command `call-home destination address http <url:port>` to point to the satellite.

Another option when direct access is not desired is to configure an HTTP or HTTPS proxy, e.g., `smart-license smart-agent proxy url https://127.0.0.1:8080`. If you plan to do this, take the note below regarding ignored CLI configurations into account:

If `ncs.conf` contains configuration for any of java-executable, java-options, override-url/url or proxy/url under the configure path `/ncs-config/smart-license/smart-agent/`, then any corresponding configuration done via the CLI is ignored.

License Registration in HA Mode

When configuring NSO in High Availability (HA) mode, the license registration token must be provided to the CLI running on the primary node. Read more about HA and node types in Chapter 7, *High Availability* in *Administration Guide*.

Licensing Log

Licensing activities are also logged in the NSO daemon log as described in the section called “Monitoring NSO” in *Administration Guide*. For example, a successful token registration results in the following log entry:

```
<INFO> 21-Apr-2016::11:29:18.022 miosaterm confd[8226]:
```

```
Smart Licensing Global Notification:
type = "notifyRegisterSuccess"
```

Check Registration Status

To check registration status, use the command `show license status`.

```
admin@ncs# show license status

Smart Licensing is ENABLED

Registration:
Status: REGISTERED
Smart Account: Network Service Orchestrator
Virtual Account: Default
Export-Controlled Functionality: Allowed
Initial Registration: SUCCEEDED on Apr 21 09:29:11 2016 UTC
Last Renewal Attempt: SUCCEEDED on Apr 21 09:29:16 2016 UTC
Next Renewal Attempt: Oct 18 09:29:16 2016 UTC
Registration Expires: Apr 21 09:26:13 2017 UTC
Export-Controlled Functionality: Allowed

License Authorization:

License Authorization:
Status: IN COMPLIANCE on Apr 21 09:29:18 2016 UTC
Last Communication Attempt: SUCCEEDED on Apr 21 09:26:30 2016 UTC
Next Communication Attempt: Apr 21 21:29:32 2016 UTC
Communication Deadline: Apr 21 09:26:13 2017 UTC
```

Modify Examples for System Install

Since all the NSO examples and README steps that come with the installer are primarily aimed at Local Install, you need to modify them in order to run them on a System Install.

To work with the System Install structure, this may require a little or bigger modification depending on the example.

For example, to port the `example.ncs/development-guide/nano-services/basic-vrouter` example to the System Install structure:

Step 1 Make the following change to the `basic-vrouter/ncs.conf` file:

```
<enabled>false</enabled>
<ip>0.0.0.0</ip>
<port>8888</port>
-<key-file>${NCS_DIR}/etc/ncs/ssl/cert/host.key</key-file>
-<cert-file>${NCS_DIR}/etc/ncs/ssl/cert/host.cert</cert-file>
+<key-file>${NCS_CONFIG_DIR}/etc/ncs/ssl/cert/host.key</key-file>
+<cert-file>${NCS_CONFIG_DIR}/etc/ncs/ssl/cert/host.cert</cert-file>
</ssl>
</transport>
```

Step 2 Copy the Local Install `$NCS_DIR/var/ncs/cdb/aaa_init.xml` file to the `basic-vrouter/` folder.

Other, more complex examples may require more `ncs.conf` file changes or require a copy of the Local Install default `$NCS_DIR/etc/ncs/ncs.conf` file together with the modification described above, or require the Local Install tool `$NCS_DIR/bin/ncs-setup` to be installed, as the `ncs-setup`

command is usually not useful with a System Install. See the section called “Migrate to System Install” for more information.

Uninstall NSO System Install

NSO can be uninstalled using the ncs-installer(1) option only if NSO is installed with `--system-install` option. Either part of the static files or full installation can be removed using `ncs-uninstall` option. Ensure to stop NSO before uninstalling.

```
# ncs-uninstall --all
```

Executing the above command removes the Installation Directory `/opt/ncs` including symbolic links, Configuration Directory `/etc/ncs`, Run Directory `/var/opt/ncs`, Log Directory `/var/log/ncs`, init scripts from `/etc/init.d` and user profile scripts from `/etc/profile.d`.

To make sure that no license entitlements are consumed after you have uninstalled NSO be sure to perform the deregister command in the CLI:

```
admin@ncs# license smart deregister
```

CHAPTER 4

Running NSO Examples

This section provides an overview of how to run the examples provided with the NSO installer. By working through the examples, the reader should get a good overview of the various aspects of NSO and hands-on experience from interacting with it.

Note This section references the examples located in `$NCS_DIR/examples.ncs`. The examples all have `README` files that includes instructions related to the example.

General Instructions

Make sure that NSO is installed with a Local Install according to the instructions in the section called "Local Install Steps". Source the `ncsrc` file in the NSO installation directory to set up a local environment. For example:

```
$ source ~/nso-6.0/ncsrc
```

Then, proceed to the example directory:

```
$ cd $NCS_DIR/examples.ncs/getting-started/using-ncs/1-simulated-cisco-ios
```

Then follow the instructions in the `README` files that are located in the example directories.

Every example directory is a complete NSO *run-time* directory. The README file and the detailed instructions later in this guide show how to generate a simulated network and NSO configuration for running the specific examples. Basically, the following steps are done:

Step 1 Create a simulated network using the **ncs-netsim --create-network** command:

```
$ ncs-netsim create-network cisco-ios-cli-3.8 3 ios
```

This creates 3 Cisco IOS devices called `ios0`, `ios1`, and `ios2`.

Step 2 Create an NSO run-time environment using the **ncs-setup** command:

```
$ ncs-setup --dest .
```

This command uses the **--dest** option to create local directories for logs, database files, and the NSO configuration file to the current directory (note that '.' refers to the current directory).

Step 3 Start ncs-netsim:

```
$ ncs-netsim start
```

Step 4 Start NSO:

```
$ ncs
```

It is important to make sure that you stop **ncs** and **ncs-netsim** when moving between examples using the **stop** option of the **netsim** and the **--stop** option of the **ncs**.

```
$ cd $NCS_DIR/examples.ncs/getting-started/1-simulated-cisco-ios
$ ncs-netsim start
$ ncs
$ ncs-netsim stop
$ ncs --stop
```

Common Mistakes

The three most common mistakes are:

1. You have not sourced the `ncsrc` file:

   ```
   $ ncs
   -bash: ncs: command not found
   ```

2. You are trying to start NSO from a directory which is not set up as a runtime directory.

   ```
   $ ncs
   Bad configuration: /etc/ncs/ncs.conf:0: "./state/packages-in-use: \
      Failed to create symlink: no such file or directory"
   Daemon died status=21
   ```

   What happens above is that NSO did not find a `ncs.conf` in the local directory so it uses the default in `/etc/ncs/ncs.conf`. That `ncs.conf` says there shall be directories at `./` such as `./state` which is not true.

   Make sure you **cd** to the "root" of the example and check that there is a `ncs.conf` file, and a `cdb-dir` directory.

3. You already have another instance of NSO running (or same with netsim):

   ```
   $ ncs
   Cannot bind to internal socket 127.0.0.1:4569 : address already in use
   Daemon died status=20
   $ ncs-netsim start
   DEVICE c0 Cannot bind to internal socket 127.0.0.1:5010 : \
     address already in use
   Daemon died status=20
   FAIL
   ```

   In order to resolve the above, just stop the running instance of NSO and netsim. And remember that it does not matter where you started the running NSO and netsim, there is no need to **cd** back to the other example before stopping.

Another problem that users run into sometimes is where the netsim device configuration are not loaded into NSO. This can happen if the order of commands are not followed. To resolve, remove the database

files in the `ncs_cdb` directory (keep any files with .xml extension). In this way NSO will reload the XML initialization files provided by **ncs-setup**.

```
$ ncs --stop
$ cd ncs-cdb/
$ ls
A.cdb
C.cdb
O.cdb
S.cdb
netsim_devices_init.xml
$ rm *.cdb
$ ncs
```

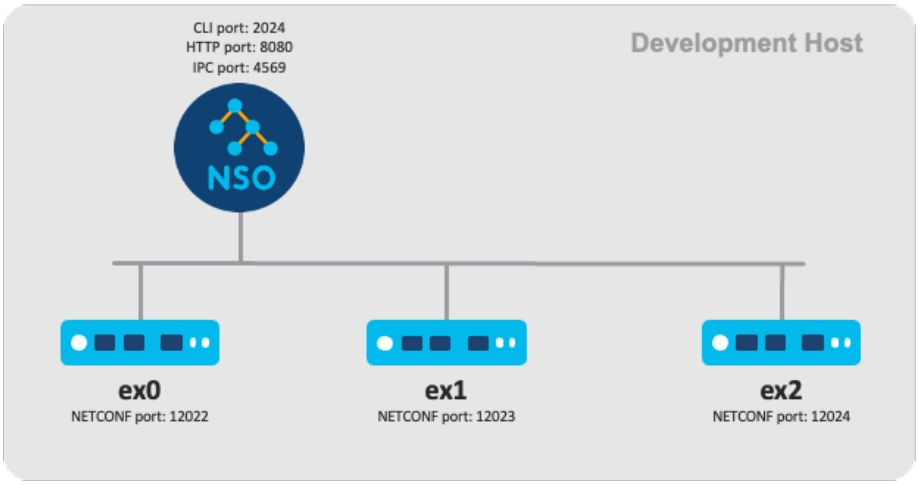
CHAPTER 5

Developing and Deploying a Nano Service

This section shows how to develop and deploy a simple NSO nano service for managing the provisioning of SSH public keys for authentication. For more details on nano services, see Chapter 31, *Nano Services for Staged Provisioning* in *Development Guide*. The example showcasing development is available under `$NCS_DIR/examples.ncs/development-guide/nano-services/netsim-sshkey`. In addition, there is a reference from the `README` in the example's directory to the deployment version of the example.

Development

Figure 8. The Development Host Topology



After installing NSO with the local-install option, development often begins with either retrieving an existing YANG model representing what the the managed network element (a virtual or physical device,

such as a router) can do or constructing a new YANG model that at least covers the configuration of interest to an NSO service. To enable NSO service development, the network element's YANG model can be used with NSO's netsim tool that uses ConfD (Configuration Daemon) to simulate the network elements and their management interfaces like NETCONF. Read more about netsim in Chapter 14, *Network Simulator* in *User Guide*.

The simple network element YANG model used for this example is available under `packages/ne/src/yang/ssh-authkey.yang`. The `ssh-authkey.yang` model implements a list of SSH public keys for identifying a user. The list of keys augments a list of users in the ConfD built-in `tailf-aaa.yang` module that ConfD uses to authenticate users.

```
module ssh-authkey {
  yang-version 1.1;
  namespace "http://example.com/ssh-authkey";
  prefix sa;

  import tailf-common {
    prefix tailf;
  }

  import tailf-aaa {
    prefix aaa;
  }

  description
    "List of SSH authorized public keys";

  revision 2023-02-02 {
    description
      "Initial revision.";
  }

  augment "/aaa:aaa/aaa:authentication/aaa:users/aaa:user" {
    list authkey {
      key pubkey-data;
      leaf pubkey-data {
        type string;
      }
    }
  }
}
```

On the network element, a Python application subscribes to ConfD to be notified of configuration changes to the user's public keys and updates the user's authorized_keys file accordingly. See `packages/ne/netsim/ssh-authkey.py` for details.

The first step is to create an NSO package from the network element YANG model. Since NSO will use NETCONF over SSH to communicate with the device, the package will be a NETCONF NED. The package can be created using the **ncs-make-package** command or the NETCONF NED builder tool. The **ncs-make-package** command is typically used when the YANG models used by the network element are available. Hence, the packages/ne package for this example was generated using the **ncs-make-package** command.

As the `ssh-authkey.yang` model augments the users list in the ConfD built-in `tailf-aaa.yang` model, NSO needs a representation of that YANG model too to build the NED. However, the service will only configure the user's public keys, so only a subset of the `tailf-aaa.yang` model that only includes the user list is sufficient. To compare, see the `packages/ne/src/yang/tailf-aaa.yang` in the example vs. the network element's version under `$NCS_DIR/netsim/confd/src/confd/aaa/tailf-aaa.yang`.

Now that the network element package is defined, next up is the service package, beginning with finding out what steps are required for NSO to authenticate with the network element using SSH public key authentication:

Step 1 First, generate private and public keys using, for example, the **ssh-keygen** OpenSSH authentication key utility.
Step 2 Distribute the public keys to the ConfD-enabled network element's list of authorized keys.
Step 3 Configure NSO to use public key authentication with the network element.
Step 4 Finally, test the public key authentication by connecting NSO with the network element.

The outline above indicates that the service will benefit from implementing several smaller (nano) steps:

- The first step only generates private and public key files with no configuration. Thus the first step should be implemented by an action before the second step runs, not as part of the second step transaction `create()` callback code configuring the network elements. The `create()` callback runs multiple times, for example, for service configuration changes, re-deploy, or commit dry-run. Therefore, generating keys should only happen when creating the service instance.
- The third step cannot execute before the second step is complete, as NSO cannot use the public key for authenticating with the network element before the network element has it in its list of authorized keys.
- The fourth step uses the NSO built-in `connect()` action and should run after the third step finishes.

What configuration input do the above steps need?

- The name of the network element that will authenticate a user with an SSH public key.
- The name of the local NSO user that maps to the remote network element user the public key authenticates.
- The name of the remote network element user.
- A passphrase used for encrypting the private key, guarding its privacy. The passphrase should be encrypted when storing it in the CDB, just like any other password.
- The name of the NSO authentication group to configure for public-key authentication with the NSO-managed network element.

A service YANG model that implements the above configuration:

```
container pubkey-dist {
  list key-auth {
    key "ne-name local-user";

    uses ncs:nano-plan-data;
    uses ncs:service-data;
    ncs:servicepoint "distkey-servicepoint";

    leaf ne-name {
      type leafref {
        path "/ncs:devices/ncs:device/ncs:name";
      }
    }
    leaf local-user {
      type leafref {
        path "/ncs:devices/ncs:authgroups/ncs:group/ncs:umap/ncs:local-user";
        require-instance false;
      }
    }
    leaf remote-name {
```

```
      type leafref {
        path "/ncs:devices/ncs:authgroups/ncs:group/ncs:umap/ncs:remote-name";
        require-instance false;
      }
      mandatory true;
    }
    leaf authgroup-name {
      type leafref {
        path "/ncs:devices/ncs:authgroups/ncs:group/ncs:name";
        require-instance false;
      }
      mandatory true;
    }
    leaf passphrase {
      // Leave unset for no passphrase
      tailf:suppress-echo true;
      type tailf:aes-256-cfb-128-encrypted-string {
        length "10..max" {
          error-message "The passphrase must be at least 10 characters long";
        }
        pattern ".*[a-z]+.*" {
          error-message "The passphrase must have at least one lower case alpha";
        }
        pattern ".*[A-Z]+.*" {
          error-message "The passphrase must have at least one upper case alpha";
        }
        pattern ".*[0-9]+.*" {
          error-message "The passphrase must have at least one digit";
        }
        pattern ".*[<>~;:!@#/$%^&*=-]+.*" {
          error-message "The passphrase must have at least one of these" +
                        " symbols: [<>~;:!@#/$%^&*=-]+";
        }
        pattern ".* .*" {
          modifier invert-match;
          error-message "The passphrase must have no spaces";
        }
      }
    }
    ...
  }
}
```

For details on the YANG statements used by the YANG model, such as `leaf`, `container`, `list`, `leafref`, `mandatory`, `length`, `pattern`, etc., see the IETF RFC 7950 that documents the YANG 1.1 Data Modeling Language. The `tailf:xyz` are YANG extension statements documented by tailf_yang_extensions(5) in *Manual Pages* .

The service configuration is implemented in YANG by a `key-auth` list where the network element and local user names are the list keys. In addition, the list has a `distkey-servicepoint` service point YANG extension statement to enable the list parameters used by the Python service callbacks that this example implements. Finally, the used `service-data` and `nano-plan-data` groupings add the common definitions for a service and the plan data needed when the service is a nano service.

For the nano service YANG part, an NSO YANG nano service behavior tree extension that references a plan outline extension implements the above steps for setting up SSH public key authentication with a network element:

```
ncs:plan-outline distkey-plan {
  description "Plan for distributing a public key";
  ncs:component-type "dk:ne" {
```

```
    ncs:state "ncs:init";
    ncs:state "dk:generated" {
      ncs:create {
        // Request the generate-keys action
        ncs:post-action-node "$SERVICE" {
          ncs:action-name "generate-keys";
          ncs:result-expr "result = 'true'";
          ncs:sync;
        }
      }
      ncs:delete {
        // Request the delete-keys action
        ncs:post-action-node "$SERVICE" {
          ncs:action-name "delete-keys";
          ncs:result-expr "result = 'true'";
        }
      }
    }
    ncs:state "dk:distributed" {
      ncs:create {
        // Invoke a Python program to distribute the authorized public key to
        // the network element
        ncs:nano-callback;
        ncs:force-commit;
      }
    }
    ncs:state "dk:configured" {
      ncs:create {
        // Invoke a Python program that in turn invokes a service template to
        // configure NSO to use public key authentication with the network
        // element
        ncs:nano-callback;
        // Request the connect action to test the public key authentication
        ncs:post-action-node "/ncs:devices/device[name=$NE-NAME]" {
          ncs:action-name "connect";
          ncs:result-expr "result = 'true'";
        }
      }
    }
    ncs:state "ncs:ready";
  }
}
ncs:service-behavior-tree distkey-servicepoint {
  description "One component per distkey behavior tree";
  ncs:plan-outline-ref "dk:distkey-plan";
  ncs:selector {
    // The network element name used with this component
    ncs:variable "NE-NAME" {
      ncs:value-expr "current()/ne-name";
    }
    // The unique component name
    ncs:variable "NAME" {
      ncs:value-expr "concat(current()/ne-name, '-', current()/local-user)";
    }
    // Component for setting up public key authentication
    ncs:create-component "$NAME" {
      ncs:component-type-ref "dk:ne";
    }
  }
}
```

The nano `service-behavior-tree` for the service point creates a nano service component for each list entry in the `key-auth` list. The last connection verification step of the nano service, the `connected` state, uses the `NE-NAME` variable. The `NAME` variable concatenates the `ne-name` and `local-user` keys from the `key-auth` list to create a unique nano service component name.

The only step that requires both a create and delete part is the `generated` state action that generates the SSH keys. If a user deletes a service instance and another network element does not currently use the generated keys, this deletes the keys too. NSO will revert the configuration automatically as part of the FASTMAP algorithm. Hence, the service list instances also need actions for generating and deleting keys.

```
  container pubkey-dist {
    list key-auth {
      key "ne-name local-user";
      ...
      action generate-keys {
        tailf:actionpoint generate-keys;
        output {
          leaf result {
            type boolean;
          }
        }
      }
      action delete-keys {
        tailf:actionpoint delete-keys;
        output {
          leaf result {
            type boolean;
          }
        }
      }
    }
  }
```

The actions have no input statements, as the input is the configuration in the service instance list entry.

The `generated` state uses the `ncs:sync` statement to ensure that the keys exist before the `distributed` state runs. Similarly, the `distributed` state uses the `force-commit` statement to commit the configuration to the NSO CDB and the network elements before the `configured` state runs.

See the `packages/distkey/src/yang/distkey.yang` YANG model for the nano service behavior tree, plan outline, and service configuration implementation.

Next, handling the key generation, distributing keys to the network element, and configuring NSO to authenticate using the keys with the network element requires some code, here written in Python, implemented by the `packages/distkey/python/distkey/distkey-app.py` script application.

The Python script application defines a Python `DistKeyApp` class specified in the `packages/distkey/package-meta-data.xml` file that NSO starts in a Python thread. This Python class inherits the `ncs.application.Application` and implement the `setup()` and `teardown()` methods. The `setup()` method registers the nano service `create()` callbacks and the action handlers for generating and deleting the key files. Using the nano service state to separate the two nano service `create()` callbacks for the distribution and NSO configuration of keys, only one Python class, the `DistKeyServiceCallbacks` class, is needed to implement them.

```
class DistKeyApp(ncs.application.Application):
    def setup(self):
        # Nano service callbacks require a registration for a service point,
        # component, and state, as specified in the corresponding data model
```

```
        # and plan outline.
        self.register_nano_service('distkey-servicepoint',  # Service point
                                   'dk:ne',                 # Component
                                   'dk:distributed',        # State
                                   DistKeyServiceCallbacks)
        self.register_nano_service('distkey-servicepoint',  # Service point
                                   'dk:ne',                 # Component
                                   'dk:configured',         # State
                                   DistKeyServiceCallbacks)

        # Side effect action that uses ssh-keygen to create the keyfiles
        self.register_action('generate-keys', GenerateActionHandler)
        # Action to delete the keys created by the generate keys action
        self.register_action('delete-keys', DeleteActionHandler)

    def teardown(self):
        self.log.info('DistKeyApp FINISHED')
```

The action for generating keys calls the OpenSSH **ssh-keygen** command to generate the private and public key files. Calling **ssh-keygen** is kept out of the service `create()` callback to avoid the key generation running multiple times, for example, for service changes, re-deploy or dry-run commits. Also, NSO encrypts the passphrase used when generating the keys for added security, see the YANG model, so the Python code decrypts it before using it with the **ssh-keygen** command.

```
class GenerateActionHandler(Action):
    @Action.action
    def cb_action(self, uinfo, name, keypath, ainput, aoutput, trans):
        '''Action callback'''
        service = ncs.maagic.get_node(trans, keypath)
        # Install the crypto keys used to decrypt the service passphrase leaf
        # as input to the key generation.
        with ncs.maapi.Maapi() as maapi:
            _maapi.install_crypto_keys(maapi.msock)
        # Decrypt the passphrase leaf for use when generating the keys
        encrypted_passphrase = service.passphrase
        decrypted_passphrase = _ncs.decrypt(str(encrypted_passphrase))
        aoutput = True
        # If it does not exist already, generate a private and public key
        if os.path.isfile(f'./{service.local_user}_ed25519') == False:
            result = subprocess.run(['ssh-keygen', '-N',
                                     f'{decrypted_passphrase}', '-t', 'ed25519',
                                     '-f', f'./{service.local_user}_ed25519'],
                                     stdout=subprocess.PIPE, check=True,
                                     encoding='utf-8')
            if "has been saved" not in result.stdout:
                aoutput = False
```

The `DeleteActionHandler` action deletes the key files if no more network elements use the user's keys:

```
class DeleteActionHandler(Action):
    @Action.action
    def cb_action(self, uinfo, name, keypath, ainput, aoutput, trans):
        '''Action callback'''
        service = ncs.maagic.get_node(trans, keypath)
        # Only delete the key files if no more network elements use this
        # user's keys
        cur = trans.cursor('/pubkey-dist/key-auth')
        remove_key = True
        while True:
            try:
                value = next(cur)
```

```
                if value[0] != service.ne_name and value[1] == service.local_user:
                    remove_key = False
                    break
            except StopIteration:
                break
        aoutput = True
        if remove_key is True:
            try:
                os.remove(f'./{service.local_user}_ed25519.pub')
                os.remove(f'./{service.local_user}_ed25519')
            except OSError as e:
                if e.errno != errno.ENOENT:
                    aoutput = False
```

The Python class for the nano service `create()` callbacks handles both the distribution and NSO configuration of the keys. The `dk:distributed` state `create()` callback code adds the public key data to the network element's list of authorized keys. For the `create()` call for the `dk:configured` state, a template is used to configure NSO to use public key authentication with the network element. The template can be called directly from the nano service, but in this case, it needs to be called from the Python code to input the current working directory to the template:

```
class DistKeyServiceCallbacks(NanoService):
    @NanoService.create
    def cb_nano_create(self, tctx, root, service, plan, component, state,
                       proplist, component_proplist):
        '''Nano service create callback'''
        if state == 'dk:distributed':
            # Distribute the public key to the network element's authorized
            # keys list
            with open(f'./{service.local_user}_ed25519.pub', 'r') as f:
                pubkey_data = f.read()
                config = root.devices.device[service.ne_name].config
                users = config.aaa.authentication.users
                users.user[service.local_user].authkey.create(pubkey_data)
        elif state == 'dk:configured':
            # Configure NSO to use a public key for authentication with
            # the network element
            template_vars = ncs.template.Variables()
            template_vars.add('CWD', os.getcwd())
            template = ncs.template.Template(service)
            template.apply('distkey-configured', template_vars)
```

The template to configure NSO to use public key authentication with the network element is available under `packages/distkey/templates/distkey-configured.xml`:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs" tags="merge">
    <authgroups>
      <group>
        <name>{authgroup-name}</name>
        <umap>
          <local-user>{local-user}</local-user>
          <remote-name>{remote-name}</remote-name>
          <public-key>
            <private-key>
              <file>
                <name>{$CWD}/{local-user}_ed25519</name>
                <passphrase>{passphrase}</passphrase>
              </file>
            </private-key>
          </public-key>
        </umap>
```

```
      </group>
    </authgroups>
    <device>
      <name>{ne-name}</name>
      <authgroup>{authgroup-name}</authgroup>
    </device>
  </devices>
</config-template>}
```

The example uses three scripts to showcase the nano service:

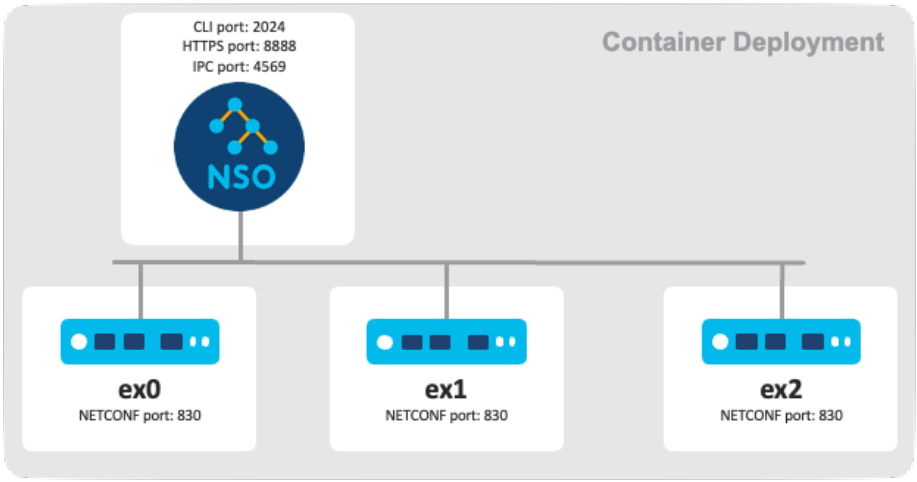
- A shell script, `showcase.sh`, that uses the **ncs_cli** program to run CLI commands via the NSO IPC port.
- A Python script, `showcase-rc.sh`, that uses the `requests` package for RESTCONF edit operations and receiving event notifications.
- A Python script that uses NSO MAAPI, `showcase-maapi.sh`, via the NSO IPC port.

The ncs_cli program identifies itself with NSO as the *admin* user without authentication, and the RESTCONF client uses plain HTTP and basic user password authentication. All three scripts demonstrate the service by generating keys, distributing the public key, and configuring NSO for public key authentication with the network elements. To run the example, see the instructions in the `README` file of the example.

Deployment

See the `README` in the `netsim-sshkey` example's directory for a reference to an NSO system installation in a container deployment variant.

Figure 9. The Deployment Container Topology



The deployment variant differs from the development example by:

- Installing NSO with a system installation for deployment instead of a local installation suitable for development

- Addressing NSO security by running NSO as the *admin* user and authenticating using public key and token.
- Rotating NSO logs to avoid running out of disk space
- Installing the `distkey` service package and `ne` NED package at startup
- The NSO CLI showcase script uses SSH with public key authentication instead of the **ncs_cli** program over unsecured IPC
- There is no Python MAAPI showcase script. Use RESTCONF over HTTPS with Python instead of Python MAAPI over unsecured IPC.
- Having NSO and the network elements (simulated by the ConfD subscriber application) run in separate containers
- NSO is either pre-installed in the NSO production container image or installed in a generic Linux container.

The deployment example sets up a minimal production installation where the NSO process runs as the *admin* OS user, relying on PAM authentication for the *admin* and *oper* NSO users. The *admin* user is authenticated over SSH using a public key for CLI and NETCONF access and over RESTCONF HTTPS using a token. The read-only *oper* user uses password authentication. The *oper* user can access the NSO WebUI over HTTPS port 443 from the container host.

A modified version of the NSO configuration file `ncs.conf` from the example running with a local install NSO is located in the `$NCS_CONFIG_DIR` (`/etc/ncs`) directory. The `packages`, `ncs-cdb`, `state`, and `scripts` directories are now under the `$NCS_RUN_DIR` (`/var/opt/ncs`) directory. The log directory is now the `$NCS_LOG_DIR` (`/var/log/ncs`) directory. Finally, the `$NCS_DIR` variable points to `/opt/ncs/current`.

Two scripts showcase the nano service:

- A shell script that runs NSO CLI commands over SSH.
- A Python script that uses the `requests` package to perform edit operations and receive event notifications.

As with the development version, both scripts will demo the service by generating keys, distributing the public key, and configuring NSO for public key authentication with the network elements.

To run the example and for more details, see the instructions in the `README` file of the deployment example.

CHAPTER 6

Supporting Information

After successful installation, to get started with NSO refer to "NSO Getting Started Guide" available with NSO Installer. Also available in `$NCS_DIR/doc/ncs_getting_started.pdf`

Refer to "NSO Users Guide" for detailed information on NSO. Available in `$NCS_DIR/doc/ncs_user_guide.pdf`

In addition to the User Guide, we recommend you to browse through the variety of useful examples found under `$NCS_DIR/examples.ncs`. Each example has a `README` file explaining its purpose and usage.

For NSO System Administration details refer to " NSO Administration Guide" available in `$NCS_DIR/doc/ncs_admin_guide.pdf`

FAQs

Below you can find few frequently asked questions or common mistakes by customers.

1. Is there a dependency between the NSO Installation Directory and Runtime Directory?
   No, there is no such dependency.
2. Do you need to source the `ncsrc` file before starting NSO?
   Local Install - Yes.
   System Install - No. (By default, the environment variables are configured and set on the shell while system install).
3. Can you start NSO from a directory, which is not a NSO runtime directory?
   Local Install - No (To start NSO, you need to point to the Run directory).
   System Install - Yes.
4. Can you stop NSO from a directory, which is not a NSO runtime directory?
   Local Install - Yes.
   System Install - Yes.
5. For evaluation and development purposes, instead of Local Install you made a System Install. Now you cannot build or run NSO examples as described in README files. How can you proceed further?
   The easiest method is to Uninstall system installation using `ncs-uninstall --all` and do a Local Install from scratch.

6. Can we move NSO Installation from one folder to another ?
   Local Install - Yes.

   You can move the directory where you installed NSO to a new location in your directory tree. Simply move NSO's root directory to the new desired location, and update this file: `$NCS_DIR/ncsrc` (and `ncsrc.tcsh` if you want) This is a small and handy script that sets up some environment variables for you. Update the paths to the new location. The `$NCS_DIR/bin/ncs` and `$NCS_DIR/bin/ncsc` scripts will determine the location of NSO's root directory automatically.
   System Install - No.

Support

- If you are evaluating NSO, you should have a designated support contact.
- If you have a NSO support agreement, please use the support channels specified in the agreement.
- In either case, please do not hesitate to send us any questions or feedback you may come up with.

Manual pages

Name

ncs-installer — NCS installation script

Synopsis

`ncs-VSN.OS.ARCH.installer.bin` [--local-install] *`LocalInstallDir`*

`ncs-VSN.OS.ARCH.installer.bin` --system-install [--install-dir *`InstallDir`*] [--config-dir *`ConfigDir`*] [--run-dir *`RunDir`*] [--log-dir *`LogDir`*] [--run-as-user *`User`*] [--keep-ncs-setup] [--non-interactive]

DESCRIPTION

The NCS installation script can be invoked to do either a simple "local installation", which is convenient for test and development purposes, or a "system installation", suitable for deployment.

LOCAL INSTALLATION

`[ --local-install ]` *`LocalInstallDir`*

When the NCS installation script is invoked with this option, or is given only the *`LocalInstallDir`* argument, NCS will be installed in the *`LocalInstallDir`* directory only.

SYSTEM INSTALLATION

`--system-install`

When the NCS installation script is invoked with this option, it will do a system installation that uses several different directories, in accordance with Unix/Linux application installation standards. The first time a system installation is done, the following actions are taken:

- The directories described below are created and populated.
- An init script for start of NCS at system boot is installed.
- User profile scripts that set up `$PATH` and other environment variables appropriately for NCS users are installed.
- A symbolic link that makes the installed version the currently active one is created (see the `--install-dir` option).

`[ --install-dir` *`InstallDir`* `]`

This is the directory where static files, primarily the code and libraries for the NCS daemon, are installed. The actual directory used for a given invocation of the installation script is *`InstallDir`*`/ncs-VSN`, allowing for coexistence of multiple installed versions. The currently active version is identified by a symbolic link *`InstallDir`*`/current` pointing to one of the `ncs-VSN` directories. If the `--install-dir` option is omitted, `/opt/ncs` will be used for *`InstallDir`*.

`[ --config-dir` *`ConfigDir`* `]`

This directory is used for config files, e.g. `ncs.conf`. If the `--config-dir` option is omitted, `/etc/ncs` will be used for *`ConfigDir`*.

`[ --run-dir` *`RunDir`* `]`

This directory is used for run-time state files, such as the CDB data base and currently used packages. If the `--run-dir` option is omitted, `/var/opt/ncs` will be used for *`RunDir`*.

`[ --log-dir LogDir ]`

This directory is used for the different log files written by NCS. If the `--log-dir` option is omitted, `/var/log/ncs` will be used for *`LogDir`*.

`[ --run-as-user User ]`

By default, the system installation will run NCS as the `root` user. If a different user is given via this option, NCS will instead be run as that user. The user will be created if it does not already exist. This mode is only supported on Linux systems that have the **setcap** command, since it is needed to give NCS components the required capabilities for some aspects of the NCS functionality.

When the option is used, the following executable files (assuming that the default `/opt/ncs` is used for `--install-dir`) will be installed with elevated privileges:

`/opt/ncs/current/lib/ncs/lib/core/pam/priv/epam`	Setuid to root. This is typically needed for PAM authentication to work with a local password file. If PAM authentication is not used, or if the local PAM configuration does not require root privileges, the setuid-root privilege can be removed by using **chmod u-s**.
`/opt/ncs/current/lib/ncs/erts/bin/ncs` `/opt/ncs/current/lib/ncs/erts/bin/ncs.smp`	Capability `cap_net_bind_service`. One of these files (normally `ncs.smp`) will be used as the NCS daemon. The files have execute access restricted to the user given via `--run-as-user`. The capability is needed to allow the daemon to bind to ports below 1024 for northbound access, e.g. port 443 for HTTPS or port 830 for NETCONF over SSH. If this functionality is not needed, the capability can be removed by using **setcap -r**.
`/opt/ncs/current/lib/ncs/bin/ip`	Capability `cap_net_admin`. This is a copy of the OS **ip(8)** command, with execute access restricted to the user given via `--run-as-user`. The program is not used by the core NCS daemon, but provided for packages that need to configure IP addresses on interfaces (such as the `tailf-hcc` package).

	If no such packages are used, the file can be removed.
`/opt/ncs/current/lib/ncs/bin/arping`	Capability `cap_net_raw`. This is a copy of the OS **arping(8)** command, with execute access restricted to the user given via `--run-as-user`. The program is not used by the core NCS daemon, but provided for packages that need to send gratuitous ARP requests (such as the `tailf-hcc` package). If no such packages are used, the file can be removed.

Note When the `--run-as-user` option is used, all OS commands executed by NCS will also run as the given user, rather than as the user specified for custom CLI commands (e.g. through clispec definitions).

`[ --keep-ncs-setup ]`	The **ncs-setup** command is not usable in a "system installation", and is therefore by default excluded from such an installation to avoid confusion. This option instructs the installation script to include **ncs-setup** in the installation despite this.
`[ --non-interactive ]`	If this option is given, the installation script will proceed with potentially disruptive changes (e.g. modifying or removing existing files) without asking for confirmation.

Name

ncs-uninstall — Command to remove NCS installation

Synopsis

`ncs-uninstall` --ncs-version [*`Version`*] [--install-dir *`InstallDir`*] [--non-interactive]

`ncs-uninstall` --all [--install-dir *`InstallDir`*] [--non-interactive]

DESCRIPTION

The **ncs-uninstall** command can be used to remove part or all of an NCS "system installation", i.e. one that was done with the `--system-install` option to the NCS installer (see ncs-installer(1)).

OPTIONS

`--ncs-version [ `*`Version`*` ]`	Removes the installation of static files for NCS version *`Version`*. I.e. the directory tree rooted at *`InstallDir`*`/ncs-`*`Version`* will be removed. The *`Version`* argument may also be given as the filename or pathname of the installation directory, or, unless `--non-interactive` is given, omitted completely in which case the command will offer selection from the installed versions.
`--all`	Completely removes the NCS installation. I.e. the whole directory tree rooted at *`InstallDir`*, as well as the directories for config files (option `--config-dir` to the installer), run-time state files (option `--run-dir` to the installer), and log files (option `--log-dir` to the installer), and also the init script and user profile scripts.
`[ --install-dir `*`InstallDir`*` ]`	Specifies the directory for installation of NCS static files, like the `--install-dir` option to the installer. If this option is omitted, `/opt/ncs` will be used for *`InstallDir`*.
`[ --non-interactive ]`	If this option is used, removal will proceed without asking for confirmation.