# Security Developer's Guide

BlackBerry | QNX

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/



**Electronic edition published: March 06, 2020**

# Contents

# About This Guide

The Security Developer's Guide is intended for system integrators who are responsible for implementing and enforcing security policies that create and maintain a trusted execution environment.

Using a wide range of evolving tactics, attackers may gain access to a system and acquire the privileges they need to take control of it. While you can't always prevent attacks, you can defend against them and reduce your loss by increasing an attacker's cost to attack.

This guide is focused on how you can design a system to defend and protect itself, thus limiting the damage to your assets and reputation resulting from an attack.

It contains best practices, examples, and refers to other documentation that supports concepts and general information in this guide.

This table may help you find what you need:

| To find out about: | See: |
|---|---|
| The importance of securing your system | "Securing Your System" in the QNX Neutrino *User's Guide* |
| Best practices for security integration | "*Best Practices*" |
| Threat models for embedded systems | "*Threat Models*" |
| Use control | • "*Rootless Execution*"<br>• "*Security Policy and Mandatory Access Control*" |
| Access control | • "*Identification and Authentication Control*"<br>• "*Sandboxing*" |
| System integrity, the secure boot process, the Merkle filesystem, and rooted chains of trust | "*Secure Boot*" |
| Levels of security for embedded systems | "*Levels of Security for Embedded Systems*" |
| The steps for developing a QNX Neutrino system that uses a security policy | "*Tutorial: Build a system that uses a security policy*" |
| The API you use to create a software backend to the `devcrypto` service. | "*The devcrypto plugin API (devcrypto_plugin.h)*" |
| An example software backend to the `devcrypto` service. | "*Example devcrypto plugin: openssl_digest.c*" |
| The `devcrypto` API that provides I/O command structures. | "*The devcrypto I/O command API (cryptodev.h)*" |

| To find out about: | See: |
|---|---|
| The `libsecpol` API that provides functions for systems that use security policies. | "*The libsecpol API (secpol.h)*" |
| Event detection | "Anomaly Detection" (the `qad` utility entry) in the *Utilities Reference* |

## Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

| Reference | Example |
|---|---|
| Code examples | `if( stream == NULL)` |
| Command options | `-lR` |
| Commands | `make` |
| Constants | NULL |
| Data types | `unsigned short` |
| Environment variables | *PATH* |
| File and pathnames | **/dev/null** |
| Function names | *exit()* |
| Keyboard chords | **Ctrl–Alt–Delete** |
| Keyboard input | `Username` |
| Keyboard keys | **Enter** |
| Program output | `login:` |
| Variable names | *stdin* |
| Parameters | *parm1* |
| User-interface components | **Navigator** |
| Window title | **Options** |

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective → Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

**CAUTION:** Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

**DANGER:** Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

**Note to Windows users**

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

## Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

# Chapter 1
# Best Practices

Follow secure coding principles. While not specific to embedded applications, a good place to start is the *general software security coding practices* maintained by the Open Web Application Security Project.

Develop and maintain a secure architecture. Build security in. Make it a default, not an option. Identify, reduce and harden attack surfaces. Favor simplicity in design to reduce the attack surface area. Take care when handling open files to reduce attack surfaces.

## Network services

Best practices for securing network services include the following specific measures:

- Keep ports closed unless keeping them open is essential to the service that the system is providing. Enable only the services and ports that are necessary.
- Move services to non-standard ports to avoid detection from broad, sweeping service scans.
- Consider using port-knocking to open debug or management ports.
- Disable `qconn` (port 8000) and `pdebug`. Some BSPs are shipped with one or both enabled.
- Avoid using FTP and Telnet services as they send user credentials in the clear. Consider using SSH and SFTP or a VPN as secure alternatives to communicate with the system.
- Keep **libssl.so** and other third-party libraries up to date.
- Disable ping response to avoid detection by ping scans. Otherwise, use filters that restrict or limit responses and provide configurable options to disable this behavior temporarily while troubleshooting network issues.
- Disable broadcast ping response to avoid an attacker using the device as part of a denial-of-service (DoS) attack.
- Review **/etc/inetd.conf** to determine which services are configured to run. Remove unnecessary services.
- Configure SSH with certificate authentication and disable password authentication. For additional security, configure a non-network backchannel (separately authenticated) to enable debug services and then use secure authentication to access it. If you are using SSH and SFTP, use keys instead of passwords where possible.
- Where possible, execute multiple `io-pkt` instances for isolating network interfaces.
- If you are using an embedded HTTP server (for example, to provide a thin client interface to an application), disable port 80 and instead use HTTPS with port 443 (its default port). If you must keep port 80 open, set it up to automatically redirect to the HTTPS site.
- Secure network endpoints.

## System services

Secure system services. Enable the **/dev/random** service, and use it to provide a source of secure random data.

### Private and public channels

If a channel doesn't need to be accessed by other processes, make it private to reduce the attack surface. Securing a channel keeps it private and reduces its vulnerability to attack.

### File systems and mount points

Best practices for securing file systems and mount points include the following:

- Pay attention to how you set ownership, file permissions, and access modes. For example, if a file is not meant to be written to, make it non-writable.
- Mark all executables as non-writable.
- Be wary of executables that have the permission bits for `setuid` (set user ID) or `setgid` (set group ID) set. While it may be a convenient way of having non-privileged processes perform privileged actions, it may also allow unintended actions.
- Many UNIX systems have mountpoints that require you to create a node or directory in an underlying filesystem before mounting a new filesystem at that point. However, QNX Neutrino does not. Don't rely on parent directory permissions to protect the mountpoint of an attached pathname space.
- Also, there is no exclusivity of mountpoints when file systems are unioned. Use channel-based mandatory access controls to protect mountpoints from registration. To protect them from client access, consider sandboxing them. Using POSIX permissions and access control lists (ACLs) further up the tree does not protect them adequately. Check the POSIX permissions and ACLs before you grant access to a client.

### Users and authentication

If SSH is enabled, consider using public key authentication instead of username- and password-based authentication. Where possible, use rootless systems. (See "*Rootless Execution*".)

### Interprocess messaging

Best practices for interprocess messaging include the following:

- Mitigate risk by authenticating messages between connected modules. Consider adding a hash or signature to the payload and use it to verify integrity.
- Statically link the processes that are system critical in **libc** to avoid situations where an agent can circumvent security features by replacing the shared library (or modified environment variables) with a different **libc**.

### Trusted execution

Best practices for trusted execution include the following:

- Embed cryptographic keys in hardware to secure the root of trust.
- Boot securely on a trusted platform.
- Use an integrity-protected filesystem (for example, the Merkle filesystem).
- Follow the principle of least privilege to ensure that every process, user, and program has access only to the information and resources that it needs. Use `procmgr` abilities, mandatory access controls, and security policy to enforce it.

## Root privilege

Minimize the need to use and keep root privilege. Minimize all root processes. Ensure that only the most essential tasks are running as **root**. Where possible, have all processes running as non-root. (See "*Rootless Execution*".)

# Chapter 2
# Threat Models

Threat modeling is the process of identifying and mitigating potential threats and risks to your system in order to reduce its vulnerability to compromise and attack. A good threat model considers each threat during the design phase of a system to ensure that appropriate countermeasures are built in.

Threats and risks to embedded systems might include system takeover by remote control or rendering a system inoperable by extracting critical data. A design that takes these risks into account reduces the attack surface and limits the damage an attack can inflict.

## Example of how to mitigate against a specific threat

Denial-of-service (DoS) can intentionally render a service or device unavailable. For example, a *smurf* attack is a type of distributed DOS attack where an attacker sends a network a series of broadcast Internet Control Message Protocol (ICMP) packets with a spoofed (disguised) return address. Many nodes on the network receive the broadcast ICMP request and respond to the spoofed return address, which is the target of the DOS attack.

This attack amplifies a single request from the attacker via the nodes on the network and results in a massive flood of responses from many systems. Your system could be participating in the attack on someone else's system even though you are not intentionally attacking it. The system being spoofed is left vulnerable to denial of service, depending on the number of responses that it receives.

To mitigate against this specific vulnerability, use the `sysctl` utility to disable a response to broadcast pings from your system. For example:

```
# sysctl -w net.inet.icmp.bmcastecho=0
```

Although it may not slow down or thwart the attacker, it does stop the amplification of attacker's broadcast requests from your node and opt you out of participating in the attack.

## Threat modeling for embedded systems

Mitigating the risks associated with embedded systems connecting to other systems should include at least the following countermeasures:

- Validating device integrity and detecting threats.
- Authenticating messages exchanged between connected modules.
- Identifying and hardening all attack surfaces.
- Following best practices.
- Staying informed about emerging vulnerabilities to embedded systems. Attackers are persistent, relentless, and creative.

## Vulnerabilities for embedded systems

The following table lists some of the known classes of embedded systems vulnerabilities and an examples of each one:

| Class | Example |
|---|---|
| Removable media | Exploited firmware update paths |
| Intermodule communication intercepts | Insufficient authorization |
| Man-in-the-middle attacks | Buffer overflows and unsafe use of functions like *strcpy()* |
| Third-party applications and malware | Improper application sandboxing and OS-level privileges |
| Compromised back-end | More traditional attack |

# Chapter 3
# Rootless Execution

Best practices for security integrators include:

- Minimize the need to use and keep root privilege.
- Minimize all root processes.
- Ensure that only the most essential tasks are running as **root**.

Rootless execution supports these best practices. In a rootless system, a process maintains its access rights and permissions. Systems are less vulnerable to subversive attacks because attackers can't obtain unrestrained access to the system.

The QNX Neutrino RTOS recognizes user ID 0, which is traditionally called **root**, as privileged. This user can do anything on the system; it has what Windows calls "administrator's privileges". Unix-style operating systems often call **root** the "superuser".

## Escalating privileges to attack a system

The user ID for **root** is zero (0). Normally, you can determine the level of privilege based on whether or not the process making the request (or the object being acted upon) is in the context of the **root** user. To escalate privileges, an attacker might look for an object that already has **root** privileges and then co-opt it for malicious purposes.

## Separating privileged operations to reduce vulnerability

In QNX Neutrino, the Process Manager ability system allows the privileged operations that are normally reserved for **root** to be granted or denied individually to any process as necessary. Processes started as **root** are, by default, granted a superset of abilities, while those not started as **root** are granted a subset.

Before rootless execution, privileged operations were restricted to **root** processes. That is, those running with UID 0.

Rootless execution allows you to use Process Manager abilities to restrict privileged operations to processes running as any user ID, provided they have been granted the specific ability that governs the specific operation.

By default, processes running as user ID 0 are granted full abilities unless:

- they are specifically restricted by dropping and locking Process Manager abilities
- they switch to a different user ID

To learn more about Process Manager (procmgr) abilities and process privileges see the “Procmgr abilities” chapter of the QNX Neutrino *Programmer's Guide*.

## Starting processes without root privilege

In practice, this method of granting privileges allows processes:

- to start with a non-root user ID and a subset of abilities by default
- to gain access to any additional abilities they require only when they require them
- to lose abilities when they are no longer required.

The ability system allows non-root processes selective access to privileged operations, providing the opportunity to deploy a system with no processes running as, or objects owned by, **root**. This approach should make it effectively impossible to achieve **root** privilege escalation. If any process is co-opted by an attacker, the operations it can be made to perform on behalf of the attacker remain limited.

# Chapter 4
# Identification and Authentication Control

Identification ensures that a user is who he or she claims to be (for example, a unique username). Authentication proves the identity of the user (for example, with a password or keys).

## Using the PAM framework

Systems that need authentication must use a configurable standard library developed for this purpose, such as a pluggable authentication module (PAM).

A PAM environment consists of the following components:

- the PAM library
- applications that link against the PAM library
- a PAM configuration file for each application
- PAM modules (DLLs) that implement the authentication and authorization mechanisms

PAM provides a framework for you to use within a system; for example, if you use username and password to authenticate users, you can use PAM to prompt for those credentials.

PAM simplifies the task of choosing which algorithm or database to use for authentication. To configure a system that has been configured to use PAM (and make use of the PAM framework), you edit a text file to specify what PAM needs to do.

The framework also simplifies the task of changing which algorithm or database to use. By editing the PAM configuration file to change which shared object file the PAM library invokes, you can change how your system authenticates users without changing its code.

The *OpenPAM* framework is integrated with QNX Neutrino to support authentication and identification for the QNX Neutrino utilities that use it.

The `login` utility is one example of a QNX Neutrino utility that is PAM-aware and supported by the PAM framework. Under the PAM framework, the `login` command is dependent on the PAM library and loads **libpam.so**. The PAM library, in turn, opens the PAM modules explicitly referenced in the PAM configuration file.

The PAM library explicitly checks permissions when it loads modules.

## Adding PAM modules

PAM modules (**pam_*.so**) are usually located in one or more of the following directories, but they can be kept elsewhere:

1. **/proc/boot**
2. **/usr/local/lib**
3. **/usr/lib**

By default, these paths are searched for PAM modules in this order.

If you plan to keep the PAM modules in a different directory, use the `confstr` value `_CS_PAMLIB` to override the default search paths and indicate where to find them. If you override the default directories, you must specify a single path. Multiple search paths are not permitted.

The following example sets a configuration string to indicate where to find the PAM modules:

```
setconf _CS_PAMLIB /system/lib/pam
```

Ensure that the permissions of the PAM module paths and the module files themselves exclude write (**w**) permission for **group** and **other**.

## Configuring PAM

Without the PAM framework in place, a login sequence typically checks **/etc/passwd** (and checks the shadow file **/etc/shadow**, if a password is set) to get a user entry and set credentials.

PAM configuration files are usually located in one or more of the following directories, but they may instead be kept elsewhere:

1. **/usr/local/etc/pam.d/**
2. **/etc/pam.d/**

By default, these paths are searched for PAM configuration files in this order.

If you plan to keep the PAM configuration files in a different directory, use the `confstr` value `_CS_PAMCONF` to override the default search paths and indicate where to find them. If you are overriding the default directories, you must specify a single path; multiple search paths are not permitted.

The following example shows how to set a configuration string to indicate where to find the PAM configuration files:

```
setconf _CS_PAMCONF /system/etc/pam.d
```

The directory may contain multiple PAM configuration files to support a range of security policies beyond authentication. For example, the `login` service would look for the configuration file **/etc/pam.d/login** or **/usr/local/etc/pam.d/login** if the default paths are used.

If no filename matches the service name (binary name) that the application registered, the service looks instead for a file named **other** in the search paths for the PAM configuration files.

Configuration commands are stacked in the PAM configuration file to create a chain. They are processed in top-down order by `libpam`. The configuration file specifies facilities, control flags, modules (shared object files) and optional arguments using the following syntax:

```
facility control_flag module arguments
```

For example:

```
auth required pam_qnx.so
auth required pam_radius.so
account required pam_permit.so
```

You specify the facility with one of the following values:

| Facility | Task | Functions |
|---|---|---|
| `account` | Account management | `pam_acct_mgmt` |
| `password` | Password management | `pam_chauthtok` |
| `auth` | Authentication | `pam_authenticate` and `pam_setcred` |
| `session` | Session management | `pam_open_session` and `pam_close_session` |

The configuration file for a service that uses PAM must specify a chain for each of the four facilities (`auth`, `account`, `session`, `password`), even if the service does not make use of a facility. If a chain is missing, *pam_start()* fails with `PAM_SYSTEM_ERR`.

The control flags describe what happens if a function of the indicated type succeeds or fails. The PAM facilities each support the following control flags:

| Flag | Purpose |
|---|---|
| `binding` | If the module fails and is part of a chain, the chain executes and the request is denied. Success breaks the chain. |
| `required` | If the module fails and is part of a chain, the chain executes and the request is denied. Success does not break the chain. |
| `requisite` | If the module fails and is part of a chain, the chain is broken. Success does not break the chain. |
| `sufficient` | Failure does not break the chain. If the module succeeds with no prior failures in the chain, the chain is broken. |
| `optional` | The result is ignored. |

For example:

```
auth sufficient pam_rootok.so
auth optional pam_motd.so nullok
auth requisite pam_qnx.so nullok
```

or

```
password requisite pam_qnx.so nullok
account requisite pam_qnx.so nullok
```

PAM attempts authentication in the order that the modules are listed, until the user is authenticated, the options for authentication are exhausted, or there is a terminal failure.

QNX Neutrino supports the following modules:

| Module | Description | Example |
|---|---|---|
| `pam_deny.so` | Always returns failure. | Permit all logins but deny any attempts to change a password:<br>`/etc/pam.d/passwd:`<br>`auth requisite pam_permit.so`<br>`account requisite pam_permit.so`<br>`session requisite pam_permit.so`<br>`password requisite pam_deny.so` |
| `pam_echo.so` | Displays a message. Fails if the message file does not exist. | Print a message from a file:<br>`/etc/pam.d/passwd:`<br>`password optional`<br>`pam_echo.so`<br>`file=/`*`path`*`/good-password.txt` |
| `pam_exec.so` | Runs a command. Fails if the command does not run. | Runs a command after each local password change:<br>`/etc/pam.d/passwd:`<br>`password optional pam_exec.so` *`/path/command`* |
| `pam_mac.so` | Determines the type that should be associated with the user and switches to it if a security policy is loaded. | Configure PAM for `ssh`, making use of the `allow_mac_policy` option (to avoid having to change any configuration based on whether a security policy is used or not):<br>`auth requisite pam_qnx.so`<br>`account requisite pam_qnx.so`<br>`session requisite pam_qnx.so`<br>`session requisite pam_mac.so allow_no_policy`<br>`password requisite pam_qnx.so` |

| Module | Description | Example |
|---|---|---|
| `pam_permit.so` | Always returns success. | Log in as any user without being prompted for a password:<br>`/etc/pam.d/login:`<br>`auth requisite pam_permit.so`<br>`account requisite pam_permit.so`<br>`session requisite pam_permit.so`<br>`password requisite pam_permit.so` |
| `pam_qnx.so` | Behavior is comparable to previous QNX versions. | Provide legacy behavior:<br>`/etc/pam.d/login:`<br>`auth requisite pam_qnx.so nullok`<br>`account requisite pam_qnx.so nullok`<br>`session requisite pam_qnx.so nullok`<br>`password requisite pam_qnx.so nullok` |
| `pam_radius.so` | Authenticate on RADIUS protocol. | Authenticate the SSH server request from the RADIUS server:<br>`/etc/pam.d/sshd:`<br>`auth sufficient pam_radius.so` |
| `pam_rootok.so` | Always returns success for the superuser. | Check for UID 0:<br>`auth sufficient pam_rootok.so` |
| `pam_self.so` | Returns success if the target user's UID matches the current UID. | Self authentication:<br>`auth requisite pam_self.so` |
| `pam_ftpusers.so` | Returns success if the user is listed in **/etc/ftpusers**. | Account management:<br>`account required pam_ftpusers.so` |
| `pam_group.so` | Accepts or rejects users based on their membership in a | Permit only members of the admin group to login:<br>`auth requisite pam_group.so`<br>`account required pam_group.so group=admin` |

| Module | Description | Example |
|---|---|---|
| | particular group. | |

For a full list of arguments for each module, see the *PAM Reference*.

If you are using these examples in a Windows environment, make sure that each line in the text file ends with a carriage return as well as a line feed.

## Utilities supported by the PAM framework

The following QNX Neutrino utilities are PAM-aware and supported by the PAM framework:

- `login`
- `ftpd`
- `pppd`
- `passwd`
- `su`
- `sshd`
- `racoon`

The `racoon` and `sshd` utilities parse their configuration files before PAM does and require additional configuration to enable PAM.

## Integrating PAM functions

The following pseudocode provides an example of how to integrate PAM functions into your system:

```
pam_start("service name", ...)
     if returns -1 go to "no PAM")
otherwise:
     auth
     acctmgmt
          chauthtok
     setcred
     opensession
     ...
     closesession
pam_end
```

The following code provides an example of how to integrate PAM functions into your system for `login`:

```
# apply additional groups as specified in groups.conf
auth optional pam_group.so
# use standard un*x validation, allow blank passwords
auth required pam_qnx.so nullok
# deny on failure
auth requisite pam_deny.so
password required pam_qnx.so nullok
password requisite pam_deny.so
account required pam_qnx.so nullok
account requisite pam_deny.so
session required pam_qnx.so nullok
session requisite pam_deny.so
```

The following code provides an example of how to integrate PAM functions into your system for `su`:

```
# root can su with no auth
auth sufficient pam_rootok.so
# use standard un*x validation, allow blank passwords
auth required pam_qnx.so nullok
# deny on failure
auth requisite pam_deny.so
account required pam_qnx.so nullok
account requisite pam_deny.so
session required pam_qnx.so nullok
session requisite pam_deny.so
# no password facility
```

## Using `syslogd` for debugging

Since OpenPAM and the PAM modules use `syslogd` to log errors, you can use the following steps to use `syslogd` for debugging:

1. Create the configuration file for syslogd: **/etc/syslog.conf** and add the following line to it:

   ```
   *.*      /dev/slog
   ```

   Use tabs in this new line, not spaces. The `syslogd` utility can't handle spaces in the configuration file and fails silently.

2. Run `slogger2` to create **/dev/slog**.
3. Run `syslogd` so that `syslogd` messages are written into **/dev/slog**.

Now you should be able to see PAM error messages using `slog2info`.

## Troubleshooting

Use this section to troubleshoot your PAM configuration.

**Invalid password database**

To troubleshoot an invalid password database, check:

- file permissions
- file ownership
- each user present in **/etc/passwd** must be in **/etc/shadow** as well and vice versa
- each group present in **/etc/passwd** must be in **/etc/group** as well

A valid set up looks like this:

| File | User | Group | Permissions |
|---|---|---|---|
| **/etc/passwd** | root | root | 0644 |
| **/etc/shadow** | root | root | 0600 |
| **/etc/group** | root | root | 0644 |

The following commands may help you find the information you need to change:

```
ls -ld /etc/passwd
ls -ld /etc/shadow
ls -ld /etc/group
cat /etc/passwd
cat /etc/shadow
cat /etc/group
```

**Incorrect permissions or ownership of utilities**

To troubleshoot incorrect permissions or ownership of utilities, check that:

- utilities are on the target
- utilities are `setuid` user
- utilities are owned by user **root** and group root

A valid set up looks like this:

| Utility | User | Group | Permissions |
|---|---|---|---|
| `login` | root | root | 4755 |
| `passwd` | root | root | 4755 |
| `su` | root | root | 4755 |
| `sshd` | root | root | 0755 |

The following commands may help you find the information that you need to change:

```
ls -ld /etc
ls -ld /etc/pam.d
ls -l /etc/pam.d
ls -ld /etc/pam.d/login
ls -ld /etc/pam.d/passwd
ls -ld /etc/pam.d/su
ls -ld /etc/pam.d/sshd
ls -ld /etc/pam.d/ftpd
ls -ld /etc/pam.d/sshd
ls -ld /etc/pam.d/racoon
grep auth /etc/pam.d/*
grep password /etc/pam.d/*
grep account /etc/pam.d/*
grep session /etc/pam.d/*
```

**Incorrect PAM configuration files**

To troubleshoot incorrect PAM configuration files, check that:

- permissions of directories are set correctly
- a PAM configuration file exists for the utility
- all four PAM chains are listed in the configuration file for the utility

A valid set up looks like this:

| Filepath | User | Group | Permissions |
|---|---|---|---|
| **/etc** | root | root | 0755 |
| **/etc/pam.d** | root | root | 0755 |
| **/etc/pam.d/*** | root | root | 0644 |

The following commands may help you find the information you need to change:

```
ls -ld /etc
ls -ld /etc/pam.d
ls -l /etc/pam.d
ls -ld /etc/pam.d/login
ls -ld /etc/pam.d/passwd
ls -ld /etc/pam.d/su
ls -ld /etc/pam.d/sshd
ls -ld /etc/pam.d/ftpd
ls -ld /etc/pam.d/sshd
ls -ld /etc/pam.d/racoon
grep auth /etc/pam.d/*
```

```
grep password /etc/pam.d/*
grep account /etc/pam.d/*
grep session /etc/pam.d/*
```

**Missing PAM modules**

To troubleshoot missing PAM modules, check that:

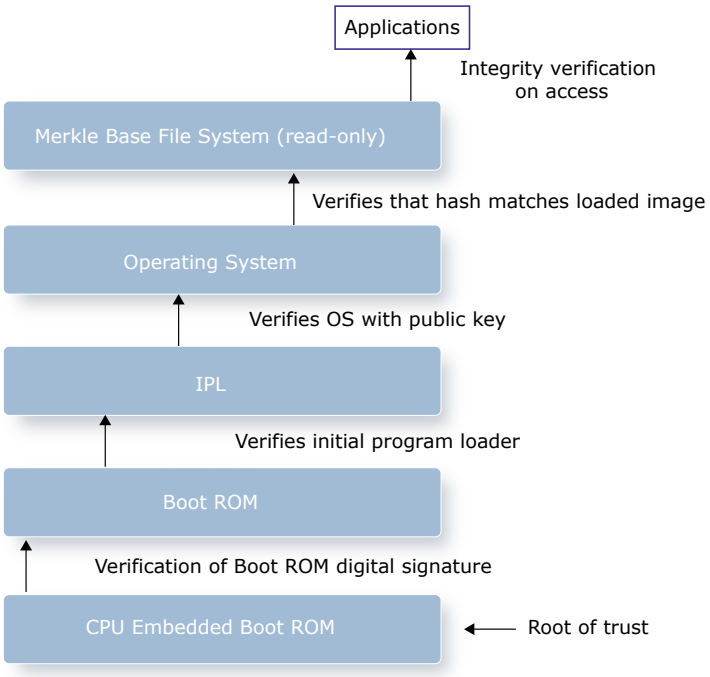
- directory permissions are set correctly
- all PAM modules are present

The following commands may help you find which modules are missing and determine which directory permissions need to change:

```
ls -ld /usr
ls -ld /usr/lib
ls -ld /usr/lib/pam_deny.so
ls -ld /usr/lib/pam_echo.so
ls -ld /usr/lib/pam_exec.so
ls -ld /usr/lib/pam_ftpusers.so
ls -ld /usr/lib/pam_group.so
ls -ld /usr/lib/pam_permit.so
ls -ld /usr/lib/pam_qnx.so
ls -ld /usr/lib/pam_radius.so
ls -ld /usr/lib/pam_rootok.so
ls -ld /usr/lib/pam_self.so
```

# Chapter 5
# Secure Boot

The secure boot mechanism is based on the concept of a chain of trust. This kind of chain is anchored (rooted), and the root (for example: a factory-blown key hash) is inherently trusted. The root of trust requires a public key that is known to the firmware. From there, a private key is used to sign files such as the IFS image.




Systematic validation during boot-up starts at the root of trust and extends to boot ROMs, primary/secondary boot loaders, IPLs, and IFS images. Each subsequent piece of firmware and software in the boot process is cryptographically validated by its predecessor.

When a securely booted system is up and running, it is considered secure unless the root of trust has been compromised. If the contents of the secured IFS image change after boot-up, the board will not reboot.

Using trusted boot impacts boot time. The magnitude of the impact depends on the size of the image and performance characteristics of the hardware.

Securing the image helps defend against low-level attacks. A trusted environment begins with a trusted platform. When secure (trusted) boot is in place, if an IFS image is compromised, it does not load or boot on the board.

QNX Trusted Disk (QTD) devices provide integrity protection of the underlying disk data via a hash tree of all filesystem blocks which are verified on demand. The root hash of the tree is signed with a key pair that provides assurance that the filesystem has not been tampered with. Accessing a part of the

filesystem that fails the integrity check returns an error. See “QNX Trusted Disk” in the *System Architecture* reference.

To learn more about how to install and build a QNX BSP, see:

- the *BSP User Guide* for your board
- *Building Embedded Systems*
- the *Release Notes* for the BSP any additional board-specific instructions

For additional information on how to secure a specific board for booting, refer to the vendor's notes for the board.

# Chapter 6
# Sandboxing

In the context of security integration, a sandbox is a mechanism for constraining code with virtual walls to protect it from being damaged by malicious code. You can also use a sandbox to limit the damage done if the code it contains is co-opted by an attacker.

Best practices for security integrators include protecting mountpoints from client access by sandboxing them. Application sandboxing (including third-party applications) can reduce vulnerability to malware. Sandboxes are also a convenient, isolated space for testing your code. Use the procmgr ability _PROCMGR_AID_SANDBOX to create and delete sandboxes, attach a process to a sandbox, and detach a process from a sandbox.

To learn more about procmgr abilities, see the "Procmgr abilities" chapter of the QNX Neutrino *Programmer's Guide*.

Child processes inherit the sandbox of their parent.

## Example: Attaching a sandbox to a process

The following example shows how to use procmgr abilities to create a sandbox, attach a process to the sandbox, and then delete the sandbox:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sandbox.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <pthread.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <sys/procmsg.h>

static const char sandbox_config[] =
        "[version=1]\n"
        "/proc/boot\n"
        "/dev/tty*\n"
        "/tmp\n"
        "/root\n"
        "/usr/lib\n"
        "!*";

int main(int argc, char *argv[])
{
   proc_sandbox_create_t   msg;
   iov_t                   iov[2];
```

```
int                     r;
int      sbid;

fprintf(stderr, "Initializing sandbox configuration\n");
r = sandbox_parse_config(sandbox_config, sizeof(sandbox_config), &iov[1], &msg.i.version);
if (r > 0) {
   fprintf(stderr, "Parse error on line %d\n", r);
   return 1;
} else if (r < 0) {
   errno = r;
   perror("sandbox_parse_config");
   return 1;
} else {
   fprintf(stderr, "%s:%d - Creating sandbox...\n", __FUNCTION__, __LINE__);
   msg.i.type = _PROC_SANDBOX;
   msg.i.subtype = _PROC_SANDBOX_CREATE;
   msg.i.datalen = iov[1].iov_len;
   iov[0].iov_base = &msg;
   iov[0].iov_len = sizeof(msg);

   if (MsgSendv(PROCMGR_COID, iov, 2, iov, 1) != 0) {
      perror("MsgSendv(_PROC_SANDBOX_CREATE)");
      return 1;
   }
   sbid = msg.o.sbid;
   sandbox_done_config(&iov[1]);
}


// Start the child process.
pid_t   pid = fork();
if (pid < 0) {
   perror("fork");
   return 1;
}

if (pid == 0) {
   delay(1000); // wait for parent to attach child to sandbox
   if (execl("/proc/boot/sh", "proc/boot/sh", NULL) < 0) {     /* execute the command  */
      printf("*** ERROR: exec failed errno:%d\n", errno);
      exit(1);
   }
   fprintf(stderr, "%s:%d - execl exitting\n", __FUNCTION__, __LINE__);
}

fprintf(stderr, "%s:%d\n", __FUNCTION__, __LINE__);
r = sandbox_attach(sbid, pid);
if (r < 0) {
   fprintf(stderr, "sandbox_attach: %s\n", strerror(-r));
   goto done;
}
fprintf(stderr, "%s:%d pid=%d r=%d\n", __FUNCTION__, __LINE__, pid, r);

waitpid(pid, &r, 0);
```

```
    fprintf(stderr, "%s:%d\n", __FUNCTION__, __LINE__);
    pid = 0;

done:
    if (sbid != -1) {
        sandbox_delete(sbid);
    }
    return r;
}
```

# Chapter 7
# Security Policy and Mandatory Access Control

A system's security policy controls where a process can attach channels in the path space, defines which abilities to assign to its processes, and controls which processes can connect to which others.

Access control is a mechanism used to secure a system by limiting the actions available to a process. *Sandboxing*, in contrast, constrains code with virtual walls (to protect it from being accessed and damaged). These security measures work well together. A security policy sets out the conditions for access and allows access when they are met.

Mandatory access control (MAC) is policy-driven, with rules to enforce relationships between processes, channels, and paths. For example, rules control which processes can connect to a channel, as well as which specific paths a process may attach to in the path space. It restricts the ability of a process to connect to a channel.

Sandboxing also provides access control but is path-based and depends on the implementation of the resource manager being accessed. Having access denied via channel-based MAC is more secure than sandboxing but is less fine grained.

There is no default security policy. To establish a security policy, you will need:

- a well-defined set of processes to secure, and information about the interfaces between them
- labels for policy enforcement (type identifiers)
- rules for enforcement (process and channel attributes)

Policy development involves defining the rules in a text file according to a strict grammar, compiling them for integration into the system, and pushing the compiled policy to `procnto` to implement them. Once in place, these mandatory access controls are enforced by the process manager rather than at the discretion of individual processes. The enforcement is type based. These rules (in the text file) yield types on individual system processes.

To establish mandatory access controls, you will need:

- a security policy
- specific  abilities for pushing policy and securing the type IDs
- the following utilities support policy development:
  - `secpol`
  - `secpolcompile`
  - `secpolgenerate`
  - `secpolmonitor`
  - `secpolpush`

## Type identifiers

A type identifier is a small integer value that is used by QNX Neutrino to define security rules and make policy-based security decisions. It is conceptually similar to a label. All security rules involve one or more types.

Type identifiers currently start at 0 and increase by one for each type declaration. The way they are numbered becomes important only when you use using Qnet and a different security policy between nodes. To ensure the correspondence of type identifiers between nodes, ensure that any types that must be used across nodes are defined in a single file that is compiled into all policies prior to any other type definitions.

In the security policy language, types are represented as names which are then converted to type identifiers in the binary policy file. Type names follow the same rules as C identifiers. They can contain alphanumeric characters and the underscore, but must not start with a digit.

You must declare all types that are used in the policy. Mandatory declaration ensures that mistyped names result in compilation errors instead of unexpected behavior at run time.

Because the type identifiers assigned to types are not fixed, code that uses types should look up the type to convert to an identifier rather than embed a number.

## Assigning types to processes and channels

When a process is assigned a type, it acquires:

- all the `procmgr` abilities associated with the type
- the ability to attach its channels to locations in the path space as defined for that type
- the ability to connect to channels of other processes as appropriate for its type.

The type associated with a process can be set at the time the process is spawned. To change the type identifier, use the `on` utility with the `-T` option.

When a channel is assigned a type, the type determines the types that other processes must have in order to connect to it.

The type identifier for a channel is initially set to equal the type identifier of the process that owns it.

Use the PROCMGR_AID_SETTYPEID ability to control the ability of a process to specify a type identifier when a new process is spawned. You can also use the PROCMGR_AID_SETTYPEID ability to indicate that a process is allowed to change its own type. Usually if a process is granted the PROCMGR_AID_SETTYPEID ability, it should be granted only a limited subrange since a process can gain the abilities of any type it is allowed to switch to or spawn processes with.

Processes started from the `ifs` startup script are type 0 by default, unless explicitly stated otherwise. They have the default set of `procmgr` abilities (the same as if no security policy is in effect) and are unaffected by abilities specified in the security policy file. They are, however, still restricted in where they can attach channels. Channels of type 0 do not restrict who may connect to them.

A process can change its own type identifier with the *procmgr_set_type_id* function. When successful, a call to this function changes the security context of the process, including `procmgr` abilities and

the paths the process may attach channels to. The `procmgr` abilities are determined by the security policy and are unaffected by abilities the process may have had prior to the call.

A process is only able to successfully make the call to *procmgr_set_type_id* if a security policy has been loaded and if the process currently possesses the `procmgr` ability PROCMGR_AID_SETTYPEID with a range that covers the type identifier that is being switched to.

## Rules for enforcement and syntax

The grammar for the text file supports syntax statements for the keywords `type`, `attribute`, `allow`, `allow_attach`, and `allow_link`. They use the following syntax:

**`type`** ***type-name*****;**

Declares a type by name. Type names may contain alphanumeric characters and underscores. They must not start with a number. Use the # character on a new line to comment on the type name. QNX recommends that you do not use hyphens in type names. The type name `default` is reserved, must be declared only if it is referenced, and is associated with type ID 0. The type name `self` is reserved, must be declared, and specifies that a process is allowed to connect to itself.

In the following example, `screen_t` is the type:

```
# Type to represent the screen process
type screen_t;
```

**`attribute`** ***attribute-name*****;**

Declares an attribute by name. Groups a set of types that have some security attributes in common. Attribute names must be different than type names. Use a comma as shown in the following syntax to associate an attribute name with a type name after you declare it:

`type` *type-name* [, *attribute-name*] ... ;

When you specify an attribute for a type, the type can do anything the attribute can do. Types can be associated with multiple attributes. Attributes do not have type IDs.

In this example, the following section of a security policy could be rewritten using an attribute and the reserved type `self`:

```
allow secure1_t secure1_t : channel connect;
allow secure2_t secure2_t : channel connect;
allow secure3_t secure3_t : channel connect;
```

For example:

```
attribute secure;
type secure1_t, secure;
type secure2_t, secure;
type secure3_t, secure;
allow secure self : channel connect;
```

**allow** ***source-set target-set*** **:** ***class-set permission-set*****;**

Grants `procmgr` abilities (or the ability to connect to channels) to a type. When a process that is a member of *source-set* acts on an object whose type is a member of *target-set* and whose class is any member of *class-set*, that process has all the permissions of *permission-set*.

There are two options for *class-set*:

- Use `ability` to bestow `procmgr` abilities on processes with a given type.
- Use `channel` to establish which channel types a process (of a given type) can connect to.

The effect of an `allow` statement is equivalent to taking all combinations of the *source-set*, *target-set*, *class-set* and *permission-set* and treating them as if that combination was used in a separate `allow` statement. The *source-set* and *target-set* can be comprised of types, attributes, or any combination of the two.

For example, the following rule uses `allow` to assign permissions for specific sources and targets:

```
allow { type1 type2 } { type3 type4 }:channel connect;
```

The following set of rules creates permissions that are equivalent to the example `allow` statement:

```
allow type1 type3:channel connect;
allow type1 type4:channel connect;
allow type2 type3:channel connect;
allow type2 type4:channel connect;
```

**The `self` type**

Use the `self` type when you need to have the target type in an allow statement match the source type. The type `self` always refers to the same type as the source type.

For example, suppose you want to allow a process of type `server1` to connect to a channel of the same type. If `server1` is a type, the following rule allows the type `server1` to connect to a channel of type `server1`

```
allow server1 self:channel connect;
```

If instead `server1` is an attribute shared by types `server_a` and `server_b`, then the statement is equivalent to the following:

```
allow server_a server_a:channel connect;
allow server_b server_b:channel connect;
```

**Using the `channel` class to control connections**

By default, a process of any type is only allowed to connect to channels that have the type `default`. But because the type of a channel defaults to the process type, channel types are usually not the `default` type. Most channels require explicit rules to allow them to be connected to.

The `channel` class supports the two following permissions:

- `connect` — governs a process's ability to connect to a channel on the same node.
- `net_connect` — governs the ability for a process from another Qnet node to access a channel on the local one.

The following `allow` statement allows a process of type `mm_client` to connect to a channel on the same node that has the type `mm_server`:

```
allow mm_client mm_server:channel connect;
```

The following statement allows this same channel to be connected to by a process of type `mm_client` that is running on another node:

```
allow mm_client mm_server:channel net_connect;
```

**Using the ability class to grant `procmgr` abilities**

Use the `allow` statement with the `ability` class to grant an ability to a type. You assign permissions using the ability names. For example, to give the type `server` the abilities that allow it to map physical memory over some range, switch to user IDs 4, 6 and 23, and create dynamic abilities, use the following rule:

```
allow server self:ability {
    mem_phys:0x1200000-0x24000000
    setuid:4,6,23
    able_create
};
```

By default, only **root** gets the abilities that are listed. To grant them to non-root as well, include the `nonroot` option:

```
allow server self:ability {
    nonroot
    mem_phys:0x1200000-0x24000000
    setuid:4,6,23
    able_create
};
```

In this rule, `nonroot` indicates that the abilities are being granted to both root and non-root. There is no facility to grant an ability to non-root only.

**Unlocking abilities and changing inheritability**

By default, the abilities are all locked and inherited. This behavior ensures abilities are fixed: if you know the type of a process, then you also know what abilities it has. If abilities are not locked or not inherited, then they are less certain. While not rendering a system insecure, it does make it harder to assess its security.

Unlocked abilities are at least partially under the control of the process. Whether it handles them well or badly is frequently unknown. Abilities that are not inherited frequently result in the loss of root abilities when a child is spawned, but can also result in an increase of abilities since subranges are discarded.

In many cases, abilities are left unlocked and not-inherited to allow the process itself to drop abilities it no longer needs after initialization and avoid having its children inherit root privileges that it required running as non-root.

An alternative to this approach is to provide the process with the type identifier that it should switch to following initialization and one or more type identifiers to spawn its children as. Then the security policy of the system can be configured centrally, outside of the individual processes.

Reasons to leave abilities unlocked or not inherited include the following:

- If a resource manager uses *procmgr_ability()* to modify its abilities and the abilities are locked, the call fails (and the resource manager may terminate).
- If the resource manager is being run as non-root with additional root abilities, then if it is given spawn or fork abilities any child process inherits the additional abilities.

To leave abilities unlocked, use the `unlock` option. To have them not-inherited, use the `noinherit` option. For example, the following rule grants the same abilities as the previous example but leaves them unlocked and not inherited:

```
allow server self:ability {
    noinherit
    unlock
    mem_phys:0x1200000-0x24000000
    setuid:4,6,23
    able_create
};
```

If any rule unlocks a particular ability it remains unlocked irrespective of any rules that may follow.

**Custom abilities**

In some cases, a process needs a custom ability (sometimes called dynamic abilities). For example, if `server` also needs to be able to bind to a privileged TCP port:

```
ability network/bind/privport;

allow server self:ability {
    network/bind/privport
    mem_phys:0x1200000-0x24000000
    setuid:4,6,23
    able_create
};
```

Custom abilities must be declared.

#### Ability ranges

As shown in some of the previous examples, when a type is granted an ability, the specification can include a range. You can specify a single range or a comma-separated list of ranges. For the majority of abilities, ranges are numeric and consist of a range start and end value separated by a hyphen. The start and end are represented in decimal, octal, or hexadecimal with conventional notation.

If only the start of range is specified with no hyphen, the start and end values are the same. If the end is omitted but the hyphen is still present, the end of the range is equal to the largest possible value.

A range for the `settypeid` and `channel_connect` abilities is a single type name because these abilities both reference types whose numeric value can change:

```
allow server self:ability {
    # Grant mem_phys over the range 1024 to 4096 and
0x1200000 to 0x24000000
    mem_phys:1024-4096,0x1200000-0x24000000
    # Grant setuid for 4, 5, 6, 23 and 96 and above
    setuid:4-6,23,96-
    # Grant settypeid for the type ids associated with types
server1 and server2
    settypeid:server1,server2
    # Grant channel_connect for type server3
    channel_connect:server3
    # Allow a switch to a type that has the pathspace ability
    gain_priv:pathspace
};
```

The ability name and any ranges may not have any space separating them. It is not necessary to specify all ranges for a given ability in a single rule. If multiple rules specify ranges for the same ability, the individual ranges are combined.

#### The `default_priv`, `root_priv`, and `nonroot_priv` pseudo-abilities

There are also several pseudo-abilities that affect which abilities a type is granted. Although pseudo-abilities make it easy to grant large sets of abilities to processes, which might makes it easier to get a system working, it also might leave the system considerably less secure. Use them with caution.

The `default_priv` pseudo-ability gives a type default behavior for all abilities not explicitly granted for the type. It changes the special PROCMGR_AID_EOL ability that terminates ability lists. Without this ability, the PROCMGR_AID_EOL ability includes the options to deny and lock all further abilities for both root and non-root. When `default_priv` is granted, no additional flags are passed with the PROCMGR_AID_EOL ability, which leaves other abilities in their default state.

The two pseudo-abilities `root_priv` and `nonroot_priv` represent the set of static abilities given by default to root and non-root processes respectively. The following two statements are equivalent:

```
allow t1 self:ability {
    root_priv
};
allow t1 self:ability {
    spawn_setuid spawn_setgid setuid setgid getid pathspace
reboot cpumode runstate confset rsrcdbmgr session
    umask event rlimit mem_add mem_phys mem_special
mem_global mem_peer mem_lock wait v86 qnet clockset
clockperiod
    interrupt keydata io trace priority connection schedule
signal timer path_trust swap child_newapp aps_root
    able_create default_timer_tolerance xprocess_query chroot
power srandom sandbox qvm rlimit_peer mac_policy settypeid
};
```

The former has the advantage because it expresses the intent more clearly. Using `root_priv` does not necessarily grant the abilities to root nor does `nonroot_priv` grant the abilities to non-root, they are simply the equivalent to listing the abilities explicitly.

For example, the following statement gives all the abilities normally granted to both root and non-root to both root and non-root users:

```
allow t1 self:ability {
    nonroot
    root_priv
    nonroot_priv
};
```

The `default_priv`, `root_priv` and `nonroot_priv` are broad in scope and provide an excessive set of abilities, but you can use a minus sign (-) to specify abilities to exclude. For example, the following statement defines a set of abilities that includes all root abilities except for `mem_phys` and `keydata`:

```
allow t1 self:ability {
    root_priv
    -mem_phys
    -keydata
};
```

When this exclusion syntax is used with `default_priv`, the type gains default treatment for all abilities it has not explicitly been granted with the exception of the ones excluded.

The minus sign does not necessarily deny the ability, though it might have this effect. The effect of negation is limited to the statement in which it is used.

In the previous example, the statement did not grant the `keydata` ability to the type because `keydata` is excluded but it is entirely possible that statements before or after might have given `keydata`.

For example, the following statements represent a complicated way of giving all root privileges to a type:

```
allow t1 self:ability {
    mem_phys
};
allow t1 self:ability {
    root_priv
    -mem_phys
    -keydata
};
allow t1 self:ability {
    keydata
};
```

**`allow_attach`** ***source-set path-set*** **[*new-type*];**

Creates a rule to let a process attach one of its own channels to a path. If the channel successfully attaches to the path, the channel optionally becomes the new type. Paths can include wildcards and ellipses.

In this example, the rule allows `io-pkt` to attach to many paths under **/dev/socket**:

```
allow_attach io_pkt_t /dev/socket/* socket_t;
```

In this next example, the rule allows `unrestricted_t` to attach a channel to any path:

```
allow_attach unrestricted_t /…;
```

**`allow_link`** ***source-set path-set*****;**

Creates a rule to let a process create a `procmgr` symbolic link or attach the channel of another process to a path. It affects the ability of a process to use the functions `pathmgr_symlink` and `pathmgr_link` (when the channel involved belongs to another process). The source of the link can be a string or a combination of node, process, and channel identifiers. Paths can include wildcards and ellipses.

In this example, the rule allows process of type `type1_t` to replace the dynamic linker by pointing it to somewhere other than **/proc/boot/libc.so.4**:

```
allow_link type1_t /usr/lib/ldqnx-64.so.2;
```

## Summary of policy rules

| Rule: | Description: |
|---|---|
| `allow` | Gives `procmgr` abilities or the ability to connect to channels to a type or attribute, or a set if you use {}. |

| Rule: | Description: |
|---|---|
| `allow_attach` | Allows the *resmgr_attach()* of a name. |
| `allow_link` | Allows the *pathmgr_symlink()* of a name. |
| `ability` | Declares a list of abilities to assign to a type or attribute. |

### Summary of special types in a policy

| Rule: | Description: |
|---|---|
| `default` | You must declare this type to use it (but you do not have to use it). Any process can connect to a channel of this type. |
| `self` | The reflexive type. When you use it as a *target-set*, it refers to the individual source types. |
| `default_rules` | The rules given to all types by default. The set of non-root `procmgr` abilities, unless you redefine this type (by declaring it and setting it differently). |

### Summary of special permissions for abilities in a policy

| Rule: | Description: |
|---|---|
| `nonroot` | Allows root-privilege ability to a non-root process. |
| `root_priv` | The default set of abilities for a root process. |
| `nonroot_priv` | The default set of abilities for a non-root process. |
| `noinherit` | The abilities that are not inherited when a process is created. |
| `unlock` | The process may change or drop abilities while running. |
| `gain_priv` | A type change may allow an increase in privilege. |

To remove an ability from a previous list, start a line with the **-** sign. Use **:range-range,range-range** (with no spaces) for subranges.

### Defining your policy

QNX recommends that the MAC security policy be developed independently by a person who is dedicated to security. The rules in this policy should take into consideration the intended *level* of system security and be used in conjunction with other security measures such as *best practices*. The person who

develops the policy does not need to be the same person who is responsible for integrating it into the system.

It can take a while to develop a security policy; it takes time to determine which abilities each specific process needs. Policy development is a scenario-dependent activity that requires trying different things with the system until you get the policy to where you want it to be.

You can use the `secpolmonitor` utility to collect the following information:

- what services the system has
- details about services that are running
- input for your policy decisions about the kinds of system activity that should and should not be allowed

The `secpolmonitor` utility is a verbose utility that also provides a way for you to determine when operations fail due to an installed security policy, usually because of an error in the policy.

## Example of an uncompiled policy file (secpol.txt)

The following example illustrates a way to use security policy to control the use of the path **/dev/screen** as is used by the screen service and its clients:

```
# Type to represent the screen process
type screen_t;
# Type to represent screen clients (you might want more than one type)
type screen_client_t;

allow_attach screen_t /dev/screen;
allow screen_client_t screen_t : channel connect;
```

This policy introduces two types named `screen_t` and `screen_client_t`, one to be used when starting the screen service, one to be used when starting its clients. The `allow_attach` rule states that any process labeled with `screen_t` is allowed to attach a channel to the path **/dev/screen**. Once the policy has been installed in `procnto`, it will not be possible to attach any path that is not explicitly allowed including `procmgr` symbolic links.

The example shows the simpler form of the `allow_attach` statement where there is no change in the type associated with the channel being attached. Channels inherit the type of the process that owns them; this means that this channel will also have a type of `screen_t`. By default, no process is able to connect to a channel with type `screen_t`. To make the channel useful, an allow statement is added to indicate that and a process with the type `screen_client_t` is allowed to connect to a channel that has a type `screen_t`.

## Working without a policy in place

If you chose not to push a policy to `procnto` (whether or not you have defined one), the type ID for all channels remains 0 and the ability of a process to attach to a path relies solely on the `procmgr` ability PROCMGR_AID_PATHSPACE.

Without a policy in place, channels retain their default type ID of 0 (since their process type ID is 0) and processes are unrestricted in their ability to connect to a channel with a type ID of 0.

### Compiling and implementing the policy

A policy can be mutable or immutable, depending on how you compile it. For more information, see the `secpolcompile` and `secpol` utilities.

### Enforcing the policy

Once the policy is pushed in place, channel-based mandatory access controls are enforced by the operating system.

### Developing a policy

The goal of writing a security policy is to be able to run processes on a system with only the capabilities they need, to minimize the potential damage that could occur should one of them be compromised. Consider that the required capabilities for many resource managers are not always fixed; for example, additional abilities may be required only during initialization. Another thing to think about is when a process spawns another one. The child process might have different security requirements from its parent process.

Trying to develop a policy from scratch, based solely on known information about interrelationships between processes and knowledge of the abilities they require, is difficult. The simplest approach is to run a system as it is intended to be run, have the `secpolmonitor` utility monitor all security events, and then you can generate a policy from its output. Running a system as it is meant to be run means starting all processes with their intended uids and types.

Taking this approach lets you directly use the `secpolmonitor` output in your policy development. For example, if it says that type `io_pkt_t` needs ability `CONFSET:19`, then you know that this type requires it (whether or not the `io-pkt` process itself, or a process it spawns, uses this ability).

Suppose, in the final system, it is intended that `io-pkt` be run as uid 21, gid 21, and type `io_pkt_t`. In the startup script for the system, you might want something like this line:

```
on -T io_pkt_t -u 21:21 io-pkt-v6-hc -de1000 -p qnet -p tcpip random
```

However, without a policy in place this won't work; since `procnto` does not know about type `io_pkt_t`, it refuses to use it. If you remove this option, then `io-pkt` fails to run because it is missing abilities. A policy needs to be loaded, but we haven't got one yet.

Write a policy that allows everything for all types, without restriction. Under this policy, the system is completely insecure and runs normally. Since everything is being run with the proper types and proper uids, the output of `secpolmonitor` should accurately reflect what's needed.

An example of a policy that can be used for policy generation follows. This policy defines four types though other types are easily added; each just needs to have the attribute `superpowers` to allow it do everything.

```
attribute superpowers;

type default_rules;
allow default_rules self:ability {};

type io_pkt_t, superpowers;
type sshd_t, superpowers;
```

```
type devb_t, superpowers;
type random_t, superpowers;

ability iofunc/dup;
ability iofunc/exec;
ability iofunc/chown;
ability iofunc/read;
ability vfs/mount-blk;
ability network/bind/privport;
ability network/interface/setpriv;

allow superpowers self:ability {
   nonroot
   default_priv
   root_priv
   nonroot_priv
   channel_connect
   iofunc/dup
   iofunc/chown
   iofunc/read
   iofunc/exec
   vfs/mount-blk
   network/interface/setpriv
   network/bind/privport
   unlock
};
```

Before running anything, you want to run the `secpolmonitor` utility with a command line such as:

```
secpolmonitor -pasno /tmp/secpol.out &
```

This command captures all path attachments, as well as uses of abilities, and records them to the file **/tmp/secpol.out**. With a policy similar to this one, the system is likely to run as it should. However, if it fails, the most likely explanation is that there are processes that create additional dynamic abilities and a process fails because it has not been granted them. For more information, look at error lines in **secpol.out**. If you find some, change the policy so that the ability names are declared in the policy file and given to the `superpowers` attribute.

By default, `secpolmonitor` treats all dynamic abilities as if they don't support subranges (since most don't and it has no means to know otherwise). If you have a dynamic ability that uses subranges and you want it captured in the policy, adjust the `secpolmonitor` command to something such as:

```
secpolmonitor -pasno /tmp/secpol.out -S subrangedAble &
```

To make the final policy as complete as possible, try and exercise as much of the system as you can while you are developing the policy. If anything is missed in this exercise, then the resulting policy

may be missing rules. It is also important to try to avoid doing things that should not be allowed. For example, if the system allows a user to log into a non-privileged account, then while the system is running this policy, the user is able to do anything (such as slay system services). If this happens while the rules are being generated, then the final rules will allow this behavior even though it is not intended. Missing rules can be corrected after the fact by having `secpolmonitor` running to detect errors.

To find rules that are unintended, look at the resulting policy file. After the system has been exercised sufficiently, the output file can be copied off the system and used as the basis of the real security policy. The simplest way to do this is to use a program to process the output. An example of such a program is `genpol`, a Python script that generates a security policy from secpolmonitor output.

One difficulty with automatic rule generation is handling processes that don't handle policies imposed upon them as expected. Processes that try to control their own abilities, for example, expecting to be started as root and then dropping abilities on their own, frequently behave differently than expected if they are constrained by a policy. Their attempt to drop abilities by calling `procmgr_ability` is liable to fail since policies generated by `secpolmonitor` typically have abilities locked.

Give the `default_priv` ability to types that are used to running this service, and make sure that the `default_rules` type has no abilities. The `genpol` script automatically does this for a number of services; add new ones by updating the `problem_cases` variable at the top of this script. Services likely to cause problems frequently show problems when you try to run them using the policy that results from this process. Alternatively, if you search for and find the string `procmgr_ability` in the binary, it is likely going to have problems.

## Summary of keywords and rules

| Category: | Syntax: |
|---|---|
| Types | `type` *type-name*;<br>The reserved type names are:<br>• `self`<br>• `default`<br>• `default_rules` |
| Attributes | `attribute` *attribute-name*; |
| Allow | `allow` *source-set target-set* : *class-set permission-set*;<br>The *class-set* contains:<br>• `channel`<br>• `ability`<br>The options for `channel` are:<br>• `connect` (most common)<br>• `net_connect` (over Qnet)<br>The options for `ability` are:<br>• `procmgr` abilities which may include ranges |

| Category: | Syntax: |
|---|---|
| | • dynamic abilities<br>• `unlock`<br>• `noinherit`<br>• pseudo-abilities<br>The pseudo-abilities are:<br>• `default_priv`<br>• `gain_priv`<br>• `root_priv`<br>• `attach`<br>• `settypeid` |
| Allow attach | `allow_attach` *source-set path-set* [*new-type*]; |
| Allow link | `allow_link` *source-set path-set*; |

# Chapter 8
# Levels of Security for Embedded Systems

This section categorizes security levels for embedded systems as follows: Critical, High, Elevated, and Guarded. While this list can help set expectations for what different levels of security might look like, it is not a comprehensive or exhaustive set of system requirements.

The QNX Neutrino RTOS has been qualified to various standards and certifications in the areas of safety and security. If you are building a safety-related system, refer to the QNX OS for Safety documentation for your release. To learn more, visit *www.qnx.com*.

## Critical Security

The following table summarizes the Critical level of embedded security:

| Consequences of compromise | Examples | Recommendations |
|---|---|---|
| • Loss of life<br>• Devasting financial and reputation losses | • Autonomous driving systems<br>• Heart defibrillator<br>• Cryptographic key storage system<br>• Train navigation system<br>• Car braking and airbag systems | • Include all mechanisms recommended for High, Elevated, and Guarded levels of system security.<br>• Follow *best practices* in line with the threat level.<br>• Securely boot with hardware key storage<br>• Establish mandatory access controls<br>• Integrity management |

## High Security

The following table summarizes the High level of embedded security:

| Consequences of compromise | Examples | Recommendations |
|---|---|---|
| • Risk to human safety<br>• Risk to important, sensitive data<br>• Large financial losses | • Car infotainment systems<br>• Physical security systems<br>• Medical monitoring devices<br>• Industrial router firmware<br>• Subsea systems | • Include all mechanisms recommended for Elevated, and Guarded levels of system security.<br>• Follow *best practices* in line with the threat level.<br>• Use an adaptive partition scheduler (APS) to ensure critical processes have the resources they need. |

| Consequences of compromise | Examples | Recommendations |
|---|---|---|
| | | • Use a high availability manager (HAM) to ensure system availability. |

## Elevated Security

The following table summarizes the Elevated level of embedded security:

| Consequences of compromise | Examples | Recommendations |
|---|---|---|
| • Increased costs<br>• Moderate loss of business, data or reputation | • Home networking systems<br>• Lighting systems<br>• Home automation systems | • Include all aspects of the Guarded level of system security.<br>• Follow *best practices* in line with the threat level.<br>• Make extensive use of POSIX permissions and access control lists (ACLs).<br>• Sign and verify boot image and critical system files at runtime.<br>• Make remote access available through port-knocking. |

## Guarded Security

The following table summarizes the Guarded level of embedded security:

| Consequences of compromise | Examples | Recommendations |
|---|---|---|
| • Negligible costs<br>• Mostly a nuisance | • Bowling score terminal<br>• Hydroponic garden control system<br>• Stereo system<br>• School projects | • Follow *best practices* in line with the threat level.<br>• Enable logging.<br>• Open only necessary network ports.<br>• Disable all debug and console ports if they are unnecessary to the system. |

# Chapter 9
# Tutorial: Build a system that uses a security policy

The steps for developing a QNX Neutrino system that uses a security policy include adding security types and generating a policy using the `secpolgenerate` utility.

## Setting up a system to use security types

Before you can generate or apply a security policy, you need to specify a type for each process in your system.

The following abbreviated startup script is for QNX Neutrino on a VMWare virtual machine and is similar to one used by many BSPs. It is the first thing that runs in a system, executed directly by the OS kernel and process manager (`procnto`) acting as a rudimentary shell:

```
[+script] startup-script = {
procmgr_symlink ../../proc/boot/libc.so /usr/lib/__LD_QNX__

slogger2
waitfor /dev/slog

# Some common servers
pipe
random -t
waitfor /dev/random

devb-eide blk cache=64M,auto=partition,vnode=2000,ncache=2000,noatime,commit=low
waitfor /dev/hd0

. . .

reopen /dev/con1

ksh /proc/boot/start-system
display_msg "---> Starting shell"
[+session] ksh -l &
}
```

To create a system that supports a security policy, you need to start every process that is still active when startup is complete with the name of a type. Currently, this type determines which abilities the process has and where in the path space it can attach itself. The simplest way to start a process with a type is to use `on` with the `-T` option. Alternatively, you can use SLM, which recognizes types.

The following startup script rewrites the example with a meaningful type name for each service. You can use any name you like, but policies are easier to work with if the name is based on the name of the associated service. Because each service has unique security requirements, you should have a different type for each service. In addition, this example uses `-U` and `-u` to start each service with a different user ID:

```
[+script] startup-script = {
    procmgr_symlink ../../proc/boot/libc.so /usr/lib/__LD_QNX__

    # Push the security policy
    secpolpush

    on -T slogger2_t slogger2 -U 20
    waitfor /dev/slog

    # Some common servers
```

```
    on -T pipe_t pipe -U 21
    on -T random_t random -t -U 22
    waitfor /dev/random

    on -T devb_t -u 23 devb-eide
        lk cache=64M,auto=partition,vnode=2000,ncache=2000,noatime,commit=low
    waitfor /dev/hd0

    . . .

    ksh /proc/boot/start-system
    echo "---> Starting shell"
    on -d -t /dev/con1 -T console_t ksh -l

    reopen /dev/con1
}
```

This revised script includes the following changes:

- It runs services with a non-root user ID two different ways. For `devb`, it starts the service with its intended user (`23`). The script starts the rest of the services as **root** and then specifies a user for the process to switch to later. Currently, QNX recommends that you run services as **root** and then switch the ID, if a service supports switching the user ID.

  You can also use user names instead of numeric user IDs, but you must make sure that the user names are present in **/etc/password**.

- It changes how it provides a shell on the console. In most cases, BSPs use the following command to provide a shell:

  ```
  [+session] ksh -l &
  ```

  The revised script uses the following command, which has a similar effect but runs the shell as its own type, `console_t`:

  ```
  on -d -t /dev/con1 -T console_t ksh -l
  ```

- It switches the system from one operating according to pre-QNX Neutrino 7.0 rules to one that supports and enforces security policies using the following command:

  ```
  secpolpush
  ```

Although these changes do all the basic work necessary to start a system using a security policy, because there is no security policy, if you try to boot you get a series of errors. For the initial steps for adding a security policy, see "*Booting the system for the first time*."

## Booting the system for the first time

After you add security types to your system, a security policy is required to boot it. The `secpolgenerate` utility provides an easy method for creating this policy.

You can build a security policy yourself by writing rules in the security policy language, but it is easier to use `secpolgenerate`. For example, the following startup script adds the lines required to start and configure `secpolgenerate`, which creates a policy, then invokes `secpolpush` to push the policy. It assumes the binaries **secpolgenerate**, **libsecpol-gen.so.1**, and **secpol-preload.so** are included in the IFS:

```
secpolgenerate -u -t 50
LD_PRELOAD=secpol-preload.so
procmgr_symlink /proc/boot/libsecpol-gen.so.1 /proc/boot/libsecpol.so.1

secpolpush
```

This code should boot your system unless there are errors in the startup script or missing files.

In this example, the system is completely insecure. The `-u` option tells `secpolgenerate` that the system should be allowed to run in unrestricted mode, which means, generally speaking, the processes' behavior is not limited in any way.

The `-t 50` option allows up to 50 security types, which is usually sufficient for a new system. However, in some cases, such as when you are starting a lot of services or the ones you were starting use many security types, a larger number of types is required.

The first time you run `secpolgenerate`, simply performing a system boot may be sufficient to generate a policy that you can use to familiarize yourself with the related security features. In most cases, you continue to exercise your system at intervals specific to your development process so that `secpolgenerate` can more thoroughly track all the possible activity that processes can generate.

## The generated security policy

The generated security policy contains a set of rules that describes the abilities that processes used and where they attached themselves in the path space, as observed by `secpolgenerate`.

The policy that `secpolgenerate` generates contains rules that cover everything that has been done on the system. Although you can attempt to exercise the system enough to create a fairly complete policy, the policy is expected to evolve over time and you can address any gaps in security later.

The security policy is a text file that contains rules in the security policy language and is located in **/dev/secpolgenerate/policy**. For description of the security policy language it uses, see "*Rules for enforcement and syntax*" in the "Security Policy and Mandatory Access Control" section.

You compile the security policy text file on your host machine using `secpolcompile`. Use one of the following methods to copy the file to your host:

- The QNX Momentics IDE **Target File System Navigator**.
- If you're running `ssh`, the `cat` command using the following format:

  ```
  ssh root@<ip-address> cat /dev/secpolgenerate/policy
  ```

You can't use `scp` to copy the policy because it copies only the length that *stat()* returns. Because `secpolgenerate` generates policy content as the file is being read, it does not provide a meaningful file size.

## Compiling the security policy

You compile the security policy provided by `secpolgenerate` using the `secpolcompile` utility.

Use the following command to compile the policy:

```
secpolcompile -o secpol.bin policy
```

For more information about this utility, see `secpolcompile` in the *Utilities Reference*.

Add the output of the compilation, **secpol.bin**, to the IFS in **/proc/boot**. To avoid having to specify the location wherever it is needed, QNX recommends that you use the default policy filepath **/proc/boot/secpol.bin**.

## Booting securely

To boot the system securely, remove the lines that run `secpolgenerate` from the startup script, rebuild the OS image, and reboot.

Make sure the compiled security policy is in the IFS in **/proc/boot**. To avoid having to specify the location wherever it is needed, QNX recommends that you use the default policy filepath **/proc/boot/secpol.bin**.

In the startup script, remove the lines that run `secpolgenerate` (described in *"Booting the system for the first time"*). In the following example, the lines are commented out:

```
# secpolgenerate -u -t 50
# LD_PRELOAD=secpol-preload.so
# procmgr_symlink /proc/boot/libsecpol-gen.so.1 /proc/boot/libsecpol.so.1

secpolpush
```

After you rebuild your OS image with these changes, you reboot the system. After the reboot, system activity is restricted to what's in the policy—behavior that `secpolgenerate` observed when you exercised the system.

# Developing systems with a security policy

In most development environments, security policies will need to be revised or updated. For example, your initial attempt at a policy may not account for all desired behavior, or you might add a new program or new version of a program. Although you can capture any new rules that are required by running `secpolgenerate` again in an unrestricted manner, in most cases it's better to selectively remove restrictions using the configuration file.

## Rerunning `secpolgenerate` in an unrestricted or selectively restricted system again

To generate new rules by rerunning the system in an unrestricted manner, in the startup script, restore the three lines that run `secpolgenerate` and are described in *"Booting the system for the first time."*

In some cases, running the system with no restrictions is not desirable because its behavior departs too far from how the system should behave when it is released, especially if you have deliberately changed the policy to restrict some processes. Instead, you can select which types to remove restrictions from. For example, you have a new version of a GPS tracking process that you want to run as type `gps_track_t`. Because you are going to issue commands over the console (which the example in "Setting up a system to use security types" starts as a shell running as type `console_t`), you want it to be unrestricted. To remove the restrictions for these two types only, create a `secpolgenerate` configuration file **secpolgenerate.cfg** that contains the following lines:

```
type gps_track_t unrestricted
type console_t unrestricted
```

Then, use the following command to start `secpolgenerate`:

```
secpolgenerate -t 0 -c conf-file-path
```

The `secpolgenerate` utility reads the existing security policy (in **/proc/boot/secpol.bin**) and then creates a policy file that identifies behavior that is not accounted for in the existing policy. In this example, `-t` specifies 0 (zero) types because the system has no new types, but specifying 50 like the earlier example would also work. The `-t` option is required because `secpolgenerate` does not modify an existing policy unless either `-t` or `-u` is used.

When you perform these steps and run a system with an existing policy, the contents of **/dev/secpolgenerate/policy** changes noticeably. Instead of containing the full security policy as before, it contains only new rules that allow any observed behavior that was not allowed by the original policy. Depending on the results and the desired system behavior, you can take some or all of these rules and apply them to your original policy. For this example, you would likely take the changes indicated for `gps_track_t` because the rules represent the requirements of a new software version. You would probably ignore rules for `console_t` because it was unrestricted simply to make development easier and it is unlikely its behavior in that context should be allowed in a system that is shipped to an end user.

## Generate additional rules by running a minimal system first

Using the configuration file to adjust how `secpolgenerate` operates is a valuable development tool. However, because `secpolgenerate` reads its configuration file early in the startup when only

the IFS is available, you have to rebuild the IFS each time you want to modify the configuration file. Rebuilding the IFS can be difficult. It is easier to have the configuration file on a writable filesystem, but in a system that uses a security policy, any processes that create a filesystem need to run as types and start after `secpolgenerate`.

For example, the following steps outline a method that overcomes this problem:

1. Determine a minimal set of resource managers that can be run to provide a writable, persistent filesystem. Because `secpolgenerate` only ever reads from its configuration file, the filesystem can initially be read-only.
2. Run this set of resource managers in a basic, insecure fashion.
3. Start `secpolgenerate` and have it read its configuration file.
4. Kill the services that were started previously and then start the system normally.

The following commands illustrate this method. In this case, because the system image has an empty **/etc/secpolgenerate.cfg**, the normal security policy is enforced. A developer having special needs would boot the system, modify **/etc/secpolgenerate.cfg** as required, and then reboot:

```
pci-server --config=/etc/pci_server.cfg
devb-eide blk alloc=demand,auto=partition,vnode=100,ncache=100,noatime,commit=low dos exe=all
waitfor /dev/hd0
mount -t qnx6 /dev/hd0t179 /
secpolgenerate -c /etc/secpolgenerate.cfg -t 0
slay devb-eide pci-server
```

Although this method slows down system start-up, it is for development purposes only and should be removed for production systems.

## The error file

The `secpolgenerate` utility provides the file **/dev/secpolgenerate/errors** to help you debug broken systems.

If you run a system with a security policy and rules are missing from it, some parts might not work. To help you troubleshoot, `secpolgenerate` captures errors in the file **/dev/secpolgenerate/errors**. For example, if you try to run `io-audio` with the type `random_t`, a type that accounts for the needs of the random process, **/dev/secpolgenerate/errors** contains the following entry:

```
allow random_t self:ability {
    io
};
```

This error indicates that the process running as type `random_t` is missing the `io` ability.

What you do with the error information is up to you. If whatever failed is something that should have worked, you can update your policy to allow it (for example, give `random_t` the `io` ability). Or, as in this case, the failure indicates something else is wrong: `io-audio` is running with the wrong type. In other cases, you can simply ignore the error.

If the system is more fundamentally broken and you cannot even access **/dev/secpolgenerate/errors**, you can also run `secpolgenerate` with the option `-v`. This option configures `secpolgenerate` to output errors to `stderr`, which allows you to see the errors even if the system does not boot sufficiently to access them by other means.

## System monitoring using `secpolgenerate`

In addition to development tasks, you can use `secpolgenerate` in a secure, deployed system to detect errors.

Ideally, `secpolgenerate` starts like any other service with its own type. When it is used for monitoring instead of policy generation, it runs after the security policy is pushed to `procnto` instead of before (that is, after `secpolpush` runs). For example, start it using the following command:

```
on -T secpolgen_t -u 40 secpolgenerate
```

When `secpolgenerate` runs this way (`on` starts `secpolgenerate` without `-t` or `-u`), it does not attempt to modify the existing policy but instead simply monitors system activity and reports any errors in **/dev/secpolgenerate/errors**.

Although this technique requires security policy rules for `secpolgen_t`, `secpolgenerate` can't provide them because it can't monitor its own usage until it starts, and after it has started it is too late to monitor its usage. However, the following set of rules is likely to work:

```
type secpolgen_t;
allow secpolgen_t self:ability {
    pathspace
    mem_phys
    prot_exec
    interrupt:2147418112
    io
    trace
    map_fixed
    public_channel
};
allow_attach secpolgen_t /dev/secpolgenerate/...;
allow_attach secpolgen_t /dev/name/local/_tracelog;
```

This set of rules will also likely provide sufficient abilities to allow a future version of `secpolgenerate` to start and to monitor itself. If it requires further abilities, the **/dev/secpolgenerate/errors** should indicate what they are.

## Security policy maintenance

Maintenance tasks for your security policy can include determining whether any generated rules should be edited or removed, manually editing policies for efficiency and simplicity, and reviewing the contents of **/dev/secpolgenerate/unused**.

The `secpolgenerate` utility is not designed to generate a security policy that you can use "as is." Rather, it is intended to provide input into a policy that you modify based on the actual needs of the system. For example, in some cases, the rules for programs like `dumper` need to change to allow it to function.

It is also possible to use the automatically generated policy only as a guide for producing policies that are equivalent to but better than ones that `secpolgenerate` produces. For example, you can use attributes in the security policy language to group related sets of capabilities and assign them to types.

In other cases, you can rewrite rules to improve them. For example, the following rule is a candidate for simplification:

```
allow_attach devc_pty_t {
    /dev/ttyp0
    /dev/ptyp0
    /dev/ttyp1
    /dev/ptyp1
    /dev/ttyp2
    /dev/ptyp2
    /dev/ttyp3
    /dev/ptyp3
    /dev/ttyp4
    /dev/ptyp4
    /dev/ttyp5
    /dev/ptyp5
    /dev/ttyp6
    /dev/ptyp6
    /dev/ttyp7
    /dev/ptyp7
};
```

This rule can be replaced with the following one:

```
allow_attach devc_pty_t {
    /dev/ttyp*
    /dev/ptyp*
};
```

When the system is exercised in new ways or as new services are added and others are taken away, you need to update the policy. To perform these updates, first use `secpolgenerate` to remove restrictions from types to allow processes to do what they like without encountering errors (see "*Developing systems with a security policy*"). Then, use the contents of **/dev/secpolgenerate/policy** to

determine which rules to add to the policy. In most cases, these additional rules are merged with existing rules rather than concatenated to the end of the existing policy file.

`secpolgenerate` also provides the file **/dev/secpolgenerate/unused**, which indicates rules or types that the system did not require. It can be used as a guide for rules or types to remove. However, it is important to consider why the rules are there because it is possible that they're only required under circumstances that have not yet occured.

## Reviewing for unnecessary rules

Although `secpolgenerate` generates policy rules based on what was needed when the system was run, these are not necessarily appropriate. QNX recommends that you review the security policy specifically to locate processes with abilities that they don't need.

The `secpolgenerate` utility gives an ability to a type under the following circumstances:

- When it observes the type use the ability. In this case, it is likely that if the process is not given the ability it will fail in some manner.
- The type makes a *procmgr_ability()* call that mentions the ability.

Although removing an ability that `secpolgenerate` has identified as necessary will likely cause whatever used it to fail, in some cases the failure can be fixed without restoring the ability. For example, you might remove the need for an ability by making changes to behavior using command-line options, a configuration file, or interaction with other programs.

Look for types that have the ability `iofunc/read`, which allows a process to read a file that POSIX permissions normally prohibit. If most or all process are run with non-root user IDs (which QNX recommends), they might acquire the `iofunc/read` ability because of errors in the file permissions configuration. In this case, their attempt to open a file should fail but was allowed by granting them a privileged ability instead. To fix this, you can remove this ability for the type, run the system, and determine whether the program indicates what failed.

## *procmgr_ability()* calls and the security policy

The **secpol-preload.so** library allows the security policy that `secpolgenerate` generates to account for resource managers that use *procmgr_ability()* to retain any abilities.

Because the use of types to control a process's security capabilities is new with QNX Neutrino 7.0, existing resource managers do not make use of the feature. Instead, many resource managers start up as **root** and then switch to a non-root user ID after calling *procmgr_ability()* to retain any abilities they need. Unfortunately, this interacts badly with the preferred approach of having abilities that are locked and inherited. If a program tries to change—or even simply mentions—an ability that is locked in a *procmgr_ability()* call, the call fails and all requested changes are ignored. In many cases, programs treat this as a fatal error and quit.

The **secpol-preload.so** library is designed to solve this problem. You are not required to use this library when you use `secpolgenerate`. However, if you don't use it, processes operate properly when you generate the policy (that is, when `secpolgenerate` runs in unrestricted mode) but fail when you apply the resulting policy. To avoid this, preload **libsecpol-preload.so** for every process. The library intercepts all *procmgr_ability()* calls and sends the details to `secpolgenerate`. These details allow `secpolgenerate` to modify the policy to accommodate any process that uses *procmgr_ability()*. The result is not optimal but is the best that can be done without modifying the affected program.

You can see when the use of *procmgr_ability()* has been detected by looking for the keywords `unlock` and `noinherit` in the generated policy. In most cases, these entries should not be changed because removing `unlock` will likely make the program fail and removing `noinherit` might allow processes to inherit elevated abilities.

QNX does not recommend that a program control its own abilities because it often does not control them optimally and there are cases of programs trying to give away abilities they need. To help avoid this problem, `secpolgenerate` annotates the abilities that it has been forced to leave unlocked with comments that indicate what the *procmgr_ability()* call was doing. For example, running `pps` generates the following rules:

```
allow pps_t self:ability {
    nonroot
    unlock
    noinherit
    # deny all lock
    spawn
    # deny all lock
    fork
    # deny all lock
    prot_exec
    # deny all lock
    pgrp
    # deny all lock
```

```
    map_fixed
};
```

These rules indicate that even if `pps` retains the abilities `spawn`, `fork`, `prot_exec`, `pgrp`, and `map_fixed`, it likely doesn't matter since `pps` denies and locks them all anyway.

For another example, `io_pkt` generates the following rules:

```
allow io_pkt_t self:ability {
    unlock
    noinherit
    # allow nonroot
    pathspace
    # allow nonroot
    mem_phys
    # allow nonroot
    keydata
    # allow nonroot
    priority
    # allow nonroot
    iofunc/read
    # allow nonroot
    iofunc/dup
    # allow nonroot
    iofunc/exec
};};
```

Unlike the set of rules generated by `pps`, this set indicates potential problems. Abilities such as `mem_phys` are quite powerful and ideally should be only be held over a limited range. Others such as `iofunc/read` might be needed but it is not immediately clear why. When processes manipulate abilities themselves via *procmgr_ability()* calls, they can easily be given abilities that they don't use. In contrast, determining abilities using `secpolgenerate` gives them abilities that they actually use.

In some cases, programs call *procmgr_ability()* and continue even if it fails (for example, `io-pkt`). It is possible to have `secpolgenerate` ignore *procmgr_ability()* calls from some types and pay attention to them for others (the equivalent of not preloading **libsecpol-preload.so.1** for those programs). To configure this behavior, add a line that uses the following format to `secpolgenerate`'s config file:

```
type io_pkt_t no_procmgr_ability
```

Although it is not guaranteed, programs should work correctly with this configuration option as long as they simply ignore the failure rather than modifying their behavior in an attempt to compensate.

## Event and state files

The `secpolgenerate` utility provides a mechanism that saves and loads data about abilities and paths that types used or failed to use.

When you run `secpolgenerate` with the `-s` option and the path to a file on a writable file system, whenever `secpolgenerate` receives a SIGTERM, before it exits, it saves in the specified file all the data related to abilities and paths that types used or failed to use. The next time you run `secpolgenerate`, it restores its previous state by reading the file.

You can also manually load files in the same format into `secpolgenerate` to simulate the use of paths and abilities.

Normally, `secpolgenerate` uses numeric IDs for types and abilities when it saves its state, which is the most efficient format. To configure `secpolgenerate` to use names for types and abilities instead, which makes the information easier to interpret, specify `-n`.

# Troubleshooting and frequently asked questions

## Troubleshooting

In general, when anything goes wrong, review the **/dev/secpolgenerate/errors** file.

**`secpolgenerate` generates new subranges each time it's run**

In some cases, especially with abilities that are subranged by virtual address, each time you run a system, `secpolgenerate` generates new rules that define new subranges for the abilities. To avoid this problem, you can remove the subranges for the affected abilities from the policy entirely and grant the ability in an unrestricted manner.

**System failures that `secpolgenerate` doesn't detect**

There are ways in which a system can fail that are invisible to `secpolgenerate` and thus won't show up in its errors file. For example, if permissions are wrong on files or devices, a resource manager that runs successfully as **root** might fail when run as non-**root**. One possible solution to this problem is to identify the paths that it fails to open and grant the resource manager access to them by adding ACLs.

## Frequently asked questions

**If I boot a system using a newly generated policy, why are some rules unused?**

Because the rules generated by `secpolgenerate` represent all the capabilities needed, if you run the system with the policy it generates, you should see no unused rules. Why isn't this always the case?

There shouldn't be many unused rules, but actions used during rule generation, such as preloading libsecpol-preload.so.1, can result in channel connections that would otherwise not occur.

**Why are errors reported when nothing fails?**

In some cases, `secpolgenerate` reports errors in **/dev/secpolgenerate/errors** that don't correspond to any failure on the system. These are an artifact of how `procnto` uses the `setuid`, `setgid`, `prot_exec`, and `priority` abilities. Some of these might be partially resolved by `secpolgenerate`, others might require `procnto` changes.

There shouldn't be many unused rules, but actions used during rule generation, such as preloading libsecpol-preload.so.1, can result in channel connections that would otherwise not occur.

- `setuid`: A process can change its effective user ID to its real user ID, saved set user ID, or any user ID granted by the `setuid` ability. However, if the change is made using the *setuid()* function, a check is made for the `setuid` ability even if the call will succeed without it. This behavior can lead to erroneous error reports and possibly the inclusion of a rule to allow `setuid` during rule generation. As of QNX Neutrino 7.0.1, this no longer occurs if *setreuid()* is used instead of *setuid()*. It's not possible to fix *setuid()* as the behavior is required to allow for proper POSIX behavior.
- `setgid`: The `setgid` ability has an issue that is similar to `setuid`.

- `prot_exec`: A process that lacks the `prot_exec` ability shows a failed ability check if it creates another thread. However, the failed check is harmless.

# Frequently asked questions

## If I boot a system using a newly generated policy, why are some rules unused?

Because the rules generated by `secpolgenerate` represent all the capabilities needed, if you run the system with the policy it generates, you should see no unused rules. Why isn't this always the case?

There shouldn't be many unused rules, but actions used during rule generation, such as preloading **libsecpol-preload.so.1**, can result in channel connections that would otherwise not occur.

## Why are errors reported when nothing fails?

In some cases, `secpolgenerate` reports errors in **/dev/secpolgenerate/errors** that don't correspond to any failure on the system. These are an artifact of how `procnto` uses the `setuid`, `setgid`, `prot_exec`, and `priority` abilities. Some of these might be partially resolved by `secpolgenerate`, others might require `procnto` changes.

- `setuid`: A process can change its effective user ID to its real user ID, saved set user ID, or any user ID granted by the `setuid` ability. However, if the change is made using the *setuid()* function, a check is made for the `setuid` ability even if the call will succeed without it. This behavior can lead to erroneous error reports and possibly the inclusion of a rule to allow `setuid` during rule generation. As of QNX Neutrino 7.0.1, this no longer occurs if *setreuid()* is used instead of *setuid()*. It's not possible to fix *setuid()* as the behavior is required to allow for proper POSIX behavior.
- `setgid`: The `setgid` ability has an issue that is similar to `setuid`.
- `prot_exec`: A process that lacks the `prot_exec` ability shows a failed ability check if it creates another thread. However, the failed check is harmless.

# Chapter 10
# The `devcrypto` plugin API (`devcrypto_plugin.h`)

The `devcrypto` plugin API allows you to create a software backend to the `devcrypto` service, which provides access to cryptographic accelerators.

For more information, see the entry for **devcrypto** in the *Utilities Reference* and *"Example `devcrypto` plugin: **openssl_digest.c**"*.

# Definitions in *devcrypto_plugin.h*

*Preprocessor macro definitions for the **devcrypto_plugin.h** header file in the **devcr** library*

## Definitions:

```
#include <dev/crypto/devcrypto_plugin.h>
```

**#define DEVCRYPTO_PLUGIN_VERSION 1**

Plugin API version.

**#define DEVCRYPTO_PLUGIN_ENTRY_POINT devcrypto_plugin_ops**

Plugin entry point name.

**#define DEVCRYPTO_PLUGIN_ENTRY_NAME "devcrypto_plugin_ops"**

Plugin entry point name as a string.

**#define DEVCRYPTO_ALG_NAME_MAX 32**

Maximum size of the plugin name, including the NUL terminator.

## Library:

**devcr**

# devcrypto_aead_cipher_op_decrypt

*Decrypt data using the specified AEAD cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_aead_cipher_op_decrypt)(devcrypto_state_ctx_t *sctx,
            const uint8_t *in,
            uint32_t insize,
            uint8_t *aad,
            uint32_t aadsize,
            uint8_t *tag,
            uint32_t tagsize,
            uint8_t *out,
            uint32_t *outsize);
```

## Arguments:

***sctx***

The state context.

***in***

The ciphertext buffer to decrypt.

***insize***

The size of the input buffer.

***aad***

The Additional Authentication Data (AAD) buffer.

***aadsize***

The size of the AAD buffer.

***tag***

The tag value to use for decryption and verification.

***tagsize***

The size of the tag buffer.

***out***

The decrypted ciphertext (plaintext).

***outsize***

The size of the output buffer and the decrypted data.

## Library:

**devcr**

## Description:

Some algorithms (e.g., AES-CCM) require the `tag` value in the initialization phase.

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_aead_cipher_op_encrypt

*Encrypt data using the specified AEAD cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_aead_cipher_op_encrypt)(devcrypto_state_ctx_t *sctx,
            const uint8_t *in,
            uint32_t insize,
            uint8_t *aad,
            uint32_t aadsize,
            uint8_t *tag,
            uint32_t tagsize,
            uint8_t *out,
            uint32_t *outsize);
```

## Arguments:

***sctx***

The state context.

***in***

The plaintext buffer to encrypt.

***insize***

The size of the input buffer.

***aad***

The Additional Authentication Data (AAD) buffer.

***aadsize***

The AAD buffer size.

***tag***

The tag value produced by encryption.

***tagsize***

The size of the tag buffer.

***out***

The encrypted plaintext (ciphertext).

***outsize***

The size of the output buffer and the encrypted data.

## Library:

**devcr**

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_aead_cipher_op_init

*Initialize an AEAD cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_aead_cipher_op_init)(devcrypto_state_ctx_t *sctx,
            const uint8_t *key,
            uint32_t keysize,
            const uint8_t *iv,
            uint32_t ivsize,
            uint8_t *tag,
            uint32_t tagsize,
            uint32_t insize,
            int encrypt);
```

## Arguments:

***sctx***

The state context.

***key***

The cipher key.

***keysize***

The size of the cipher key.

***iv***

The initialization vector (IV) buffer.

***ivsize***

The size of the IV buffer.

***tag***

The tag value (only used for decryption).

***tagsize***

The size of the tag buffer.

***insize***

The size of the input data to encrypt or decrypt.

***encrypt***

Either 1 (encrypt function) or 0 (decrypt function).

## Library:

**devcr**

## Description:

Because devcrypto works using ciphers with no padding, plugins need to make sure that cipher padding is turned off. The caller is responsible for providing the cipher with padded and aligned data.

The insize argument is only used for algorithms that require that the plaintext size be known before encryption or decryption can begin (e.g., AES-CCM). This requirement also means the algorithm must encrypt or decrypt all the data in a single operation.

The tag argument is only used for algorithms that require the tag to be input before decryption can begin (e.g., AES-CCM).

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_aead_cipher_ops_t

*AEAD cipher algorithm functions*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_aead_cipher_ops {
    devcrypto_aead_cipher_op_init init;
    devcrypto_aead_cipher_op_encrypt encrypt;
    devcrypto_aead_cipher_op_decrypt decrypt;
} devcrypto_aead_cipher_ops_t;
```

## Data:

**devcrypto_aead_cipher_op_init** ***init***

The cipher initialization function.

**devcrypto_aead_cipher_op_encrypt** ***encrypt***

The cipher encryption function.

**devcrypto_aead_cipher_op_decrypt** ***decrypt***

The cipher decryption function.

## Library:

**devcr**

# devcrypto_aead_cipher_params_t

*AEAD cipher algorithm parameters*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_aead_cipher_params {
    uint32_t minkeysize;
    uint32_t maxkeysize;
    uint32_t ivsize;
    uint32_t mintagsize;
    uint32_t maxtagsize;
    uint32_t blocksize;
} devcrypto_aead_cipher_params_t;
```

## Data:

**uint32_t *minkeysize***

The minimum key size in bytes.

**uint32_t *maxkeysize***

The maximum key size in bytes.

**uint32_t *ivsize***

The default initialization vector (IV) size in bytes.

**uint32_t *mintagsize***

The minimum tag size in bytes.

**uint32_t *maxtagsize***

The maximum tag size in bytes.

**uint32_t *blocksize***

The size of the cipher block in bytes.

## Library:

**devcr**

# devcrypto_aead_cipher_t

*AEAD cipher algorithm object*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_aead_cipher {
    devcrypto_aead_cipher_params_t params;
    devcrypto_aead_cipher_ops_t ops;
} devcrypto_aead_cipher_t;
```

## Data:

**devcrypto_aead_cipher_params_t *params***

The authenticated encryption with associated data (AEAD) cipher parameters.

**devcrypto_aead_cipher_ops_t *ops***

The AEAD cipher functions.

## Library:

**devcr**

# devcrypto_algorithm_op_init

*Initialize an algorithm*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_algorithm_op_init)(devcrypto_state_ctx_t *sctx);
```

## Arguments:

***sctx***

The state context.

## Library:

**devcr**

## Description:

This function is called when an algorithm is requested to allow plugins to allocate resources and prepare the algorithm for use.

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_algorithm_op_uninit

*Uninitialize an algorithm*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef void(* devcrypto_algorithm_op_uninit)(devcrypto_state_ctx_t *sctx);
```

## Arguments:

***sctx***

The state context.

## Library:

**devcr**

## Description:

This function signals that the user has released the algorithm object and the plugin can clean up the resources associated with it.

# devcrypto_algorithm_t

*Algorithm object*

## Synopsis:

```
#include <SPECIFY HEADER LOCATION/devcrypto_plugin.h>

typedef struct _devcrypto_algorithm {
    const char name[DEVCRYPTO_ALG_NAME_MAX];
    uint32_t type;
    devcrypto_algorithm_type_t
    union {
        devcrypto_digest_t digest;
        devcrypto_cipher_t cipher;
        devcrypto_aead_cipher_t aead_cipher;
        devcrypto_mac_t mac;
    }
    devcrypto_algorithm_op_init init;
    devcrypto_algorithm_op_uninit uninit;
    void* extra;
} devcrypto_algorithm_t;
```

## Data:

**const char *name*[DEVCRYPTO_ALG_NAME_MAX]**

The algorithm name.

**uint32_t *type***

The algorithm type. See *devcrypto_algorithm_type_t*.

**devcrypto_digest_t *digest***

The digest algorithm object.

**devcrypto_cipher_t *cipher***

The cipher algorithm object.

**devcrypto_aead_cipher_t *aead_cipher***

The AEAD cipher algorithm object.

**devcrypto_mac_t *mac***

The MAC algorithm object.

**devcrypto_algorithm_op_init** *init*

The algorithm initialization function.

**devcrypto_algorithm_op_uninit** *uninit*

The algorithm un-initialization function.

**void*** *extra*

Extra storage for plugin use.

## Library:

**devcr**

# devcrypto_algorithm_type_t

*Algorithm types*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef enum {
    DEVCRYPTO_UNKNOWN_TYPE = 0,
    DEVCRYPTO_DIGEST_TYPE,
    DEVCRYPTO_CIPHER_TYPE,
    DEVCRYPTO_AEAD_CIPHER_TYPE,
    DEVCRYPTO_MAC_TYPE
} devcrypto_algorithm_type_t;
```

## Data:

**DEVCRYPTO_UNKNOWN_TYPE**

Unknown algorithm.

**DEVCRYPTO_DIGEST_TYPE**

Digest algorithm.

**DEVCRYPTO_CIPHER_TYPE**

Cipher algorithm.

**DEVCRYPTO_AEAD_CIPHER_TYPE**

AEAD cipher algorithm.

**DEVCRYPTO_MAC_TYPE**

MAC algorithm.

## Library:

**devcr**

# devcrypto_cipher_op_decrypt

*Decrypt data using the specified cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_cipher_op_decrypt)(devcrypto_state_ctx_t *sctx,
            const uint8_t *in,
            uint32_t insize,
            uint8_t *out,
            uint32_t *outsize);
```

## Arguments:

***sctx***

The state context.

***in***

The ciphertext to decrypt.

***insize***

The size of the input buffer.

***out***

The decrypted ciphertext (plaintext).

***outsize***

The size of the output buffer and decrypted data.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_cipher_op_encrypt

*Encrypt data using the specified cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_cipher_op_encrypt)(devcrypto_state_ctx_t *sctx,
            const uint8_t *in,
            uint32_t insize,
            uint8_t *out,
            uint32_t *outsize);
```

## Arguments:

***sctx***

The state context.

***in***

The plaintext to encrypt.

***insize***

The size of the input buffer.

***out***

The encrypted plaintext (ciphertext).

***outsize***

The size of the output buffer and encrypted data.

## Library:

**devcr**

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_cipher_op_init

*Initialize a cipher algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_cipher_op_init)(devcrypto_state_ctx_t *sctx,
            const uint8_t *key,
            uint32_t keysize,
            const uint8_t *iv,
            uint32_t ivsize,
            int encrypt);
```

## Arguments:

***sctx***

The state context.

***key***

The cipher key.

***keysize***

The cipher key size.

***iv***

The initialization vector (IV).

***ivsize***

The IV size.

***encrypt***

Either `1` (encrypt function) or `0` (decrypt function).

## Library:

**devcr**

## Description:

Because `devcrypto` works using ciphers with no padding, plugins need to make sure that cipher padding is turned off. The caller is responsible for providing the cipher with padded and aligned data.

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_cipher_ops_t

*Cipher algorithm functions*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_cipher_ops {
    devcrypto_cipher_op_init init;
    devcrypto_cipher_op_encrypt encrypt;
    devcrypto_cipher_op_decrypt decrypt;
} devcrypto_cipher_ops_t;
```

## Data:

**devcrypto_cipher_op_init** ***init***

The cipher initialization function.

**devcrypto_cipher_op_encrypt** ***encrypt***

The cipher encryption function.

**devcrypto_cipher_op_decrypt** ***decrypt***

The cipher decryption function.

## Library:

**devcr**

# devcrypto_cipher_params_t

*Cipher algorithm parameters*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_cipher_params {
    uint32_t minkeysize;
    uint32_t maxkeysize;
    uint32_t ivsize;
    uint32_t blocksize;
} devcrypto_cipher_params_t;
```

## Data:

**uint32_t *minkeysize***

The minimum key size in bytes.

**uint32_t *maxkeysize***

The maximum key size in bytes.

**uint32_t *ivsize***

The default initialization vector (IV) size in bytes.

**uint32_t *blocksize***

The cipher block size in bytes.

## Library:

**devcr**

# devcrypto_cipher_t

*Cipher algorithm object*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_cipher {
    devcrypto_cipher_params_t params;
    devcrypto_cipher_ops_t ops;
} devcrypto_cipher_t;
```

## Data:

**devcrypto_cipher_params_t *params***

The cipher parameters.

**devcrypto_cipher_ops_t *ops***

The cipher functions.

## Library:

**devcr**

# devcrypto_digest_op_copy

*Copy a digest algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_digest_op_copy)(devcrypto_state_ctx_t *ssctx,
            devcrypto_state_ctx_t *dsctx);
```

## Arguments:

***ssctx***

The source state context.

***dsctx***

The destination state context.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_digest_op_final

*Finalize a digest algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_digest_op_final)(devcrypto_state_ctx_t *sctx,
            uint8_t *digest,
            uint32_t *size);
```

## Arguments:

***sctx***

The state context.

***digest***

The digest buffer.

***size***

The size of the digest buffer and the final digest.

## Library:

**devcr**

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_digest_op_init

*Initialize a digest algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_digest_op_init)(devcrypto_state_ctx_t *sctx);
```

## Arguments:

***sctx***

The state context.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_digest_op_update

*Update a digest algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_digest_op_update)(devcrypto_state_ctx_t *sctx,
            const uint8_t *data,
            uint32_t size);
```

## Arguments:

***sctx***

The state context.

***data***

The data buffer to digest.

***size***

The data buffer size.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_digest_ops_t

*Digest algorithm functions*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_digest_ops {
    devcrypto_digest_op_init init;
    devcrypto_digest_op_update update;
    devcrypto_digest_op_final final;
    devcrypto_digest_op_copy copy;
} devcrypto_digest_ops_t;
```

## Data:

**devcrypto_digest_op_init** ***init***

The digest initialization function.

**devcrypto_digest_op_update** ***update***

The digest update function.

**devcrypto_digest_op_final** ***final***

The digest finalization function.

**devcrypto_digest_op_copy** ***copy***

The digest copy state function.

## Library:

**devcr**

# devcrypto_digest_params_t

*Digest algorithm parameters*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_digest_params {
    uint32_t digestsize;
    uint32_t blocksize;
} devcrypto_digest_params_t;
```

## Data:

**uint32_t *digestsize***

The digest size in bytes.

**uint32_t *blocksize***

The block size in bytes.

## Library:

**devcr**

# devcrypto_digest_t

*Digest algorithm object*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_digest {
    devcrypto_digest_params_t params;
    devcrypto_digest_ops_t ops;
} devcrypto_digest_t;
```

## Data:

**devcrypto_digest_params_t *params***

The digest parameters.

**devcrypto_digest_ops_t *ops***

The digest functions.

## Library:

**devcr**

# devcrypto_mac_op_final

*Finalize a MAC algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_mac_op_final)(devcrypto_state_ctx_t *sctx,
            uint8_t *mac,
            uint32_t *size);
```

## Arguments:

***sctx***

The state context.

***mac***

The MAC buffer.

***size***

The size of the MAC buffer and final MAC.

## Library:

**devcr**

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_mac_op_init

*Initialize a MAC algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_mac_op_init)(devcrypto_state_ctx_t *sctx,
            const uint8_t *key,
            uint32_t keysize);
```

## Arguments:

***sctx***

The state context.

***key***

The MAC key buffer.

***keysize***

The size of the MAC key buffer.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_mac_op_update

*Update a MAC algorithm state*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_mac_op_update)(devcrypto_state_ctx_t *sctx,
            const uint8_t *data,
            uint32_t size);
```

## Arguments:

***sctx***

The state context.

***data***

The data buffer to digest.

***size***

The data buffer size.

## Library:

**devcr**

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_mac_ops_t

*MAC algorithm functions*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_mac_ops {
    devcrypto_mac_op_init init;
    devcrypto_mac_op_update update;
    devcrypto_mac_op_final final;
} devcrypto_mac_ops_t;
```

## Data:

**devcrypto_mac_op_init** ***init***

The MAC initialization function.

**devcrypto_mac_op_update** ***update***

The MAC encryption function.

**devcrypto_mac_op_final** ***final***

The MAC finalization function.

## Library:

**devcr**

# devcrypto_mac_params_t

*MAC algorithm parameters*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_mac_params {
    uint32_t tagsize;
} devcrypto_mac_params_t;
```

## Data:

**uint32_t *tagsize***

The MAC tag size in bytes.

## Library:

**devcr**

# devcrypto_mac_t

*MAC algorithm object*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_mac {
    devcrypto_mac_params_t params;
    devcrypto_mac_ops_t ops;
} devcrypto_mac_t;
```

## Data:

**devcrypto_mac_params_t *params***

The MAC parameters.

**devcrypto_mac_ops_t *ops***

The MAC functions.

## Library:

**devcr**

# devcrypto_plugin_op_init

*Initialize a plugin*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef int(* devcrypto_plugin_op_init)(const char *opts);
```

## Arguments:

***opts***

The option string.

## Library:

**devcr**

## Description:

This function is called to initialize the plugin and any resources that it requires. If the plugin supports options, they are passed to it as a string using the devcrypto command line option -o (option).

## Returns:

EOK if successful or errno if an error occurred.

# devcrypto_plugin_op_uninit

*Uninitialize a plugin*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef void(* devcrypto_plugin_op_uninit)(void);
```

## Library:

**devcr**

# devcrypto_plugin_ops_t

*Plugin global functions*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_plugin_ops {
    uint16_t version;
    devcrypto_plugin_op_init init;
    devcrypto_plugin_op_uninit uninit;
} devcrypto_plugin_ops_t;
```

## Data:

**uint16_t** ***version***

The plugin version. Always set to DEVCRYPTO_PLUGIN_VERSION.

**devcrypto_plugin_op_init** ***init***

The plugin initialization function.

**devcrypto_plugin_op_uninit** ***uninit***

The plugin uninitialization function.

## Library:

**devcr**

## Description:

The devcrypto service calls these functions when it initializes or uninitializes a plugin.

# devcrypto_plugin_register_algorithm()

*Register an algorithm*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

int devcrypto_plugin_register_algorithm(const devcrypto_algorithm_t *alg)
```

## Arguments:

***alg***

The algorithm object.

## Library:

**devcr**

## Returns:

EOK if successful or `errno` if an error occurred.

# devcrypto_state_ctx_t

*Algorithm state context*

## Synopsis:

```
#include <dev/crypto/devcrypto_plugin.h>

typedef struct _devcrypto_state_ctx {
    void* data;
    const devcrypto_algorithm_t* alg;
} devcrypto_state_ctx_t;
```

## Data:

**void*** ***data***

Algorithm-specific context data.

**const devcrypto_algorithm_t*** ***alg***

The algorithm associated with this context.

## Library:

**devcr**

## Description:

The state context object is passed as a parameter to each algorithm's functions to allow algorithm-specific data to be stored and referenced. Additionally, it provides a reference to the algorithm object, which allows that object to be used.

# Chapter 11
# Example `devcrypto` plugin: `openssl_digest.c`

This example illustrates how to implement a `devcrypto` plugin.

For more information, see *"The `devcrypto` plugin API (**devcrypto_plugin.h**)"* and the entry for **devcrypto** in the *Utilities Reference*.

```
/*
 * $QNXLicenseC:
 * Copyright 2018, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#include <errno.h>
#include <stdlib.h>

#include <crypto/devcrypto_plugin.h>

#include <openssl/evp.h>

#include "openssl.h"

/*
 * Openssl digest context
 */
typedef struct _digest_ctx
{
    const EVP_MD    *md;       /* digest algorithm */
    EVP_MD_CTX      *mdctx;    /* digest context */
} digest_ctx;

/*
 * Openssl digest algorithm initialization
 *
 * @sctx: state context object
 * @alg: algorithm object
 *
 * Return:
```

```
 * - EOK on success
 * - errno on failure
 */
static int devcrypto_openssl_digest_alg_init(devcrypto_state_ctx_t *sctx)
{
 int ret = EOK;
 digest_ctx *ctx;

 ctx = calloc(1, sizeof(*ctx));
 if (!ctx) {
  ret = ENOMEM;
  goto done;
 }

 ctx->md = EVP_get_digestbyname(sctx->alg->name);
 if (!ctx->md) {
  ret = ELIBBAD;
  goto err;
 }

 ctx->mdctx = EVP_MD_CTX_create();
 if (!ctx->mdctx) {
  ret = ENOMEM;
  goto err;
 }

 sctx->data = ctx;
 ret = EOK;

 goto done;

err:
 free(ctx);

done:
 return ret;
}

/*
 * Openssl digest algorithm un-initialization
 *
 * @sctx: state context object
 *
 * Return:
 * - EOK on success
 */
static void devcrypto_openssl_digest_alg_uninit(devcrypto_state_ctx_t *sctx)
{
 digest_ctx *ctx = sctx->data;

 EVP_MD_CTX_destroy(ctx->mdctx);
 free(ctx);
}
```

```
/*
 * Openssl Digest Init
 *
 * @sctx: state context object
 *
 * Return:
 * - EOK on success
 * - errno on failure
 */
static int devcrypto_openssl_digest_init(devcrypto_state_ctx_t *sctx)
{
 int ret;
 digest_ctx *ctx = sctx->data;

 ret = EVP_DigestInit(ctx->mdctx, ctx->md);
 if (ret <= 0) {
  return ELIBBAD;
 }
 return EOK;
}

/*
 * Openssl Digest Update
 *
 * @sctx: state context object
 * @data: binary data to hash in digest
 * @size: size of @data
 *
 * Return:
 * - EOK on success
 * - errno on failure
 */
static int devcrypto_openssl_digest_update(devcrypto_state_ctx_t *sctx, const uint8_t *data, uint32_t size)
{
 int ret;
 digest_ctx *ctx = sctx->data;

 ret = EVP_DigestUpdate(ctx->mdctx, data, size);
 if (ret <= 0) {
  return ELIBBAD;
 }
 return EOK;
}

/*
 * Openssl Digest Init
 *
 * @sctx: state context object
 * @digest: buffer to store digest value
 * @size: pointer to store digest size
 *
 * Return:
 * - EOK on success
 * - errno on failure
```

```
 */
static int devcrypto_openssl_digest_final(devcrypto_state_ctx_t *sctx, uint8_t *digest, uint32_t *size)
{
 int ret;
 digest_ctx *ctx = sctx->data;
 unsigned int dsize;

 ret = EVP_DigestFinal(ctx->mdctx, digest, &dsize);
 if (ret <= 0) {
  return ELIBBAD;
 }

 if (size) {
  *size = (uint32_t)dsize;
 }

 return EOK;
}

/* Algorithms */

/*
 * Digests
 */
#define OPENSSL_DIGEST_ALG(dgstname, dgst, DGST)        \
static const devcrypto_algorithm_t dgst = {             \
    .name =    dgstname,                                \
    .type =    CRYPTO_ ## DGST,                         \
    .init =    devcrypto_openssl_digest_alg_init,       \
    .uninit =  devcrypto_openssl_digest_alg_uninit,     \
    .digest = {                                         \
        .params = {                                     \
            .digestsize =   DGST ## _DIGEST_SIZE,       \
            .blocksize =    DGST ## _BLOCK_SIZE,        \
        },                                              \
        .ops = {                                        \
            .init =     devcrypto_openssl_digest_init,  \
            .update =   devcrypto_openssl_digest_update,\
            .final =    devcrypto_openssl_digest_final, \
        },                                              \
    },                                                  \
};

/* Define algorithms */
OPENSSL_DIGEST_ALG("md5",md5,MD5)
OPENSSL_DIGEST_ALG("sha1",sha1,SHA1)
OPENSSL_DIGEST_ALG("sha224",sha224,SHA224)
OPENSSL_DIGEST_ALG("sha256",sha256,SHA256)
OPENSSL_DIGEST_ALG("sha384",sha384,SHA384)
OPENSSL_DIGEST_ALG("sha512",sha512,SHA512)
OPENSSL_DIGEST_ALG("sha512-224",sha512_224,SHA512_224)
OPENSSL_DIGEST_ALG("sha512-256",sha512_256,SHA512_256)

/*
```

```
 * TODO
 */
void devcrypto_openssl_digest_register(void)
{
 devcrypto_plugin_register_algorithm(&md5);
 devcrypto_plugin_register_algorithm(&sha1);
 devcrypto_plugin_register_algorithm(&sha224);
 devcrypto_plugin_register_algorithm(&sha256);
 devcrypto_plugin_register_algorithm(&sha384);
 devcrypto_plugin_register_algorithm(&sha512);
 devcrypto_plugin_register_algorithm(&sha512_224);
 devcrypto_plugin_register_algorithm(&sha512_256);
}
```

# Chapter 12
# The `devcrypto` I/O command API (`cryptodev.h`)

The `devcrypto` API that provides I/O command structures.

The `devcrypto` service I/O command API is derived from **libc** and the I/O command structures that **cryptodev.h** provides.

To use the `devcrypto` I/O commands, clients also need to link to **libdevcr-ioctl.so**.

For more information, see the entry for **devcrypto** in the *Utilities Reference*.

# General definitions in *cryptodev.h*

*Preprocessor macro definitions for the **cryptodev.h** header file in the **devcr** library*

## Definitions:

**#define AALG_MAX_RESULT_LEN 64**

Maximum algorithm result length.

**#define HASH_MAX_LEN AALG_MAX_RESULT_LEN**

Maximum hash algorithm result length.

**#define CRYPTO_HMAC_MAX_KEY_LEN 512**

Maximum HMAC key length.

**#define CRYPTO_CIPHER_MAX_KEY_LEN 64**

Maximum cipher key length.

**#define CRIOGET_NOT_NEEDED 1**

CRIOGET command file descriptor duplicate flag. This flag indicates that to use the cryptography device for operations, a driver doesn't require the caller to call `ioctl(fd, CRIOGET, &dupfd)` to get an extra file descriptor.

## Library:

**devcr**

# Cryptography device algorithms

Algorithms supported by the cryptography device.

## Description:

See *cryptodev_session_op_t*.

## Definitions:

```
#define CRYPTO_INVALID 0

#define CRYPTO_DES_CBC 1

#define CRYPTO_3DES_CBC 2

#define CRYPTO_BLF_CBC 3

#define CRYPTO_CAST_CBC 4

#define CRYPTO_SKIPJACK_CBC 5

#define CRYPTO_MD5_HMAC 6

#define CRYPTO_SHA1_HMAC 7

#define CRYPTO_RIPEMD160_HMAC 8

#define CRYPTO_MD5_KPDK 9

#define CRYPTO_SHA1_KPDK 10

#define CRYPTO_RIJNDAEL128_CBC 11

#define CRYPTO_AES_CBC CRYPTO_RIJNDAEL128_CBC

#define CRYPTO_ARC4 12

#define CRYPTO_MD5 13

#define CRYPTO_SHA1 14

#define CRYPTO_SHA2_256_HMAC 15

#define CRYPTO_SHA2_HMAC CRYPTO_SHA2_256_HMAC
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA256_HMAC CRYPTO_SHA2_256_HMAC
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_NULL_HMAC 16
#define CRYPTO_NULL_CBC 17
#define CRYPTO_DEFLATE_COMP 18
#define CRYPTO_MD5_HMAC_96 19
#define CRYPTO_SHA1_HMAC_96 20
#define CRYPTO_RIPEMD160_HMAC_96 21
#define CRYPTO_GZIP_COMP 22
#define CRYPTO_DEFLATE_COMP_NOGROW 23
#define CRYPTO_SHA2_384_HMAC 24
#define CRYPTO_SHA384_HMAC CRYPTO_SHA2_384_HMAC
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA2_512_HMAC 25
#define CRYPTO_SHA512_HMAC CRYPTO_SHA2_512_HMAC
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_CAMELLIA_CBC 26
#define CRYPTO_AES_CTR 27
#define CRYPTO_AES_XCBC_MAC_96 28
#define CRYPTO_AES_XCBC_MAC CRYPTO_AES_XCBC_MAC_96
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_AES_GCM_16 29
#define CRYPTO_AES_128_GMAC 30
#define CRYPTO_AES_192_GMAC 31
#define CRYPTO_AES_256_GMAC 32
#define CRYPTO_AES_GMAC 33
#define CRYPTO_SHA2_224 34
#define CRYPTO_SHA224 CRYPTO_SHA2_224
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA2_256 35

#define CRYPTO_SHA256 CRYPTO_SHA2_256
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA2_384 36

#define CRYPTO_SHA384 CRYPTO_SHA2_384
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA2_512 37

#define CRYPTO_SHA512 CRYPTO_SHA2_512
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_SHA2_224_HMAC 38

#define CRYPTO_SHA224_HMAC CRYPTO_SHA2_224_HMAC
```

Defines an alternative algorithm name to allow backwards compatibility.

```
#define CRYPTO_AES_XTS 39

#define CRYPTO_AES_ECB 40

#define CRYPTO_AES_GCM 41

#define CRYPTO_SHA512_224 42

#define CRYPTO_SHA512_256 43

#define CRYPTO_SHA512_224_HMAC 44

#define CRYPTO_SHA512_256_HMAC 45

#define CRYPTO_ALGORITHM_ALL 46

#define CRYPTO_ALGORITHM_MIN (CRYPTO_INVALID + 1)

#define CRYPTO_ALGORITHM_MAX (CRYPTO_ALGORITHM_ALL - 1)
```

# Cryptography device operations

Operations executed using ciphers.

## Description:

See *cryptodev_crypt_op_t*.

## Definitions:

**#define COP_ENCRYPT 0x0**

Encrypt operation.

**#define COP_DECRYPT 0x1**

Decrypt operation.

# Cryptography device CIOCCRYPT command flags

Flags that affect how the cryptography operation is handled.

## Description:

Only digest or MAC operations support the COP_FLAG_UPDATE and COP_FLAG_FINAL flags. See *cryptodev_crypt_op_t*.

## Definitions:

**#define COP_FLAG_NONE (0 << 0)**

No flag.

**#define COP_FLAG_UPDATE (1 << 0)**

Update to hash in a series of updates.

**#define COP_FLAG_FINAL (1 << 1)**

Final update to hash in a series of updates.

# cphash_op_t

*CIOCCPHASH command*

## Synopsis:

```
#include <dev/crypto/cryptodev.h>

typedef struct cphash_op {
    uint64_t dst_ses;
    uint64_t src_ses;
} cphash_op_t;
```

## Data:

**uint64_t *dst_ses***

The destination session.

**uint64_t *src_ses***

The source session.

## Library:

**devcr**

## Description:

This command copies a digest state from one session to another to continue the digest operation.

# cryptodev_crypt_auth_op_t

*CIOCAUTHCRYPT command*

## Synopsis:

```
#include <dev/crypto/cryptodev.h>

typedef struct crypt_auth_op {
    uint64_t ses;
    uint16_t op;
    uint16_t flags;
    uint32_t len;
    uint32_t auth_len;
    uint8_t* auth_src;
    uint8_t* src;
    uint8_t* dst;
    uint8_t* tag;
    uint32_t tag_len;
    uint8_t* iv;
    uint32_t iv_len;
} cryptodev_crypt_auth_op_t;
```

## Data:

**uint64_t *ses***

The session identifier. See *cryptodev_session_op_t*.

**uint16_t *op***

COP_ENCRYPT or COP_DECRYPT. See *Cryptography device operations*.

**uint16_t *flags***

Reserved for future use.

**uint32_t *len***

The input or output data length.

**uint32_t *auth_len***

The length of the authenticated data.

**uint8_t* *auth_src***

Additional authenticated data.

**uint8_t*** ***src***

The input data buffer.

**uint8_t*** ***dst***

The output data buffer.

**uint8_t*** ***tag***

The tag input or output buffer.

**uint32_t** ***tag_len***

The tag length.

**uint8_t*** ***iv***

The initialization vector (IV).

**uint32_t** ***iv_len***

The IV length.

## Library:

**devcr**

## Description:

This command handles AEAD ciphers such as AES-GCM.

# cryptodev_crypt_op_t

*CIOCCRYPT command*

## Synopsis:

```
#include <dev/crypto/cryptodev.h>

struct crypt_op {
    uint64_t ses;
    uint16_t op;
    uint16_t flags;
    uint32_t len;
    uint8_t* src;
    uint8_t* dst;
    uint8_t* mac;
    uint8_t* iv;
} cryptodev_crypt_op_t;
```

## Data:

**uint64_t *ses***

The session identifier. See *cryptodev_session_op_t*.

**uint16_t *op***

COP_ENCRYPT or COP_DECRYPT. See *Cryptography device operations*.

**uint16_t *flags***

See *Cryptography device CIOCCRYPT command flags*.

**uint32_t *len***

The input or output data length.

**uint8_t* *src***

The input data buffer.

**uint8_t* *dst***

The output data buffer.

**uint8_t* *mac***

The hash or MAC output buffer.

**uint8_t*** ***iv***

The initialization vector for the encryption or decryption operation.

## Library:

**devcr**

## Description:

This command is used for the following cryptography operations:

- Ciphers (except AEAD, which is handled by *cryptodev_crypt_auth_op_t*)
- Cipher and Message Authenticated Code (MAC) combinations
- Digests
- MAC

# cryptodev_session_op_t

*CIOCGSESSION command*

## Synopsis:

```
#include <dev/crypto/cryptodev.h>

struct session_op {
    uint32_t cipher;
    uint32_t mac;
    uint32_t keylen;
    uint8_t* key;
    uint32_t mackeylen;
    uint8_t* mackey;
    uint64_t ses;
} cryptodev_session_op_t;
```

## Data:

**uint32_t *cipher***

The cipher algorithm. See *Cryptography device algorithms*.

**uint32_t *mac***

The MAC algorithm. See *Cryptography device algorithms*.

**uint32_t *keylen***

The cipher key length.

**uint8_t* *key***

The cipher key.

**uint32_t *mackeylen***

The MAC key length.

**uint8_t* *mackey***

The MAC key.

**uint64_t *ses***

The session identifier.

## Library:

**devcr**

## Description:

This command is used to initiate a session with devcrypto by specifying the algorithm a client wants to use.

If devcrypto doesn't support this algorithm, the *ioctl()* call returns -1 and errno is set to ENOTSUP. Any other error means that the cryptography device encountered another error internally.

Only specific combinations of MAC and ciphers are valid.

# Chapter 13
# The `libsecpol` API (`secpol.h`)

The `libsecpol` library API provides functions for systems that use security policies.

You can use the `libsecpol` functions to perform the following tasks:

- Easily reduce privileges after initialization.
- Spawn child processes with reduced privileges.
- Perform fine-grained permission checking.
- Translate type names to type IDs and vice versa.

You can easily design a program that uses `libsecpol` to work both with and without security policies.

## Specifying the security policy file handle

For convenience, the API functions that specify a handle to the security policy file (except for *secpol_close()*) allow you to specify NULL for *handle*.

If you are calling multiple functions, you can first call `secpol_open(NULL, SECPOL_USE_AS_DEFAULT)`, which specifies that the system's default security policy file is used and that file is used when *handle* is NULL for the subsequent functions.

When you call a single function, you can omit a call to *secpol_open()* that sets a default security policy. Instead, set *handle* for the function to NULL, which will specify the default system security policy file.

## Customizing permissions using a security policy

The *secpol_get_permission()* and *secpol_check_permission()* functions allow you to support fine-grained permission checking in a system that uses security policies.

You can use *secpol_check_permission()* in one of the following two ways, which depend on how the function's *otype* (object type) parameter is used:

- Check a permission based on both the type of the requester and the operation being performed (for example, to restrict one or more options in a set of filesystem mount operations)
- Check a permission based on the object being acted on as well as the type of the requester and the operation being performed (for example, to restrict the filesystem mount options allowed for a specific mountpoint)

### Permission checking by operation

To check a permission based on both the type of the requester and the operation being performed, set *otype* to the type ID of the resource manager itself (obtained by calling *secpol_type_id()*).

For example, while currently QNX systems use a single ability to control mounting a filesystem, it could be changed to use security policy permissions instead. The following security policy entries define a client who can only mount and unmount filesystems (`fs_client_t`) and a super client who can perform all operations (`fs_super_client_t`):

```
class fs_operation { mount unmount suid trusted before after };

allow fs_client_t filesystem_t:fs_operation {
    mount unmount
};
allow fs_super_client_t filesystem_t:fs_operation {
    mount unmount suid trusted before after
};
```

This configuration does not restrict permissions based on the object that the operation is performed on. If the resource manager supports multiple filesystems, the same restrictions apply to all of them.

The filesystem resource manager runs with a type of `filesystem_t`. To get the permission handles that *secpol_check_permission()* requires, it calls *secpol_get_permission()* with the appropriate class and permission. For example:

```
s_mount_perm = secpol_get_permission(NULL, "fs_operation", "mount", 0);
```

The following code example illustrates how the filesystem can check permissions for a mount request:

```
if ( secpol_check_permission(&ctp->info, secpol_type_id(), s_mount_perm) != 0 ) {
    return EPERM;
}
```

### Permission checking by operation and object acted on

You can extend the fine-grained security illustrated in the previous example to additionally check the object being acted on (i.e., the mountpoint).

For example, a system has filesystems mounted in several places. One mountpoint must be carefully protected because it is used to store binaries and other files for system services. A second mountpoint is used to mount user-supplied USB thumb drives. Although it is likely not important to control who can mount things on the second mountpoint, any filesystems mounted there should not honor setuid bits (which would allow a binary to execute with a user ID equal to the owner of the file, including executing as **root**).

The security file to support this example defines `system_fs_t` and `user_fs_t` to represent the two mountpoints:

```
type system_fs_t;
type user_fs_t;

class fs_operation {
    mount unmount suid trusted before after
};

allow {
    fs_client_t fs_super_client_t
} user_fs_t:fs_operation {
    mount unmount
};
allow fs_super_client_t  system_fs_t:fs_operation {
    mount unmount suid trusted before after
};
```

Both regular and super clients can perform mount and unmount operations on the mountpoint for the USB drive (configured to be treated as type `user_fs_t`) but any setuid bit is disregarded. The super client can perform any operation on the system mountpoint (configured to be treated as `system_fs_t`).

The filesystem resource manager (again running as `filesystem_t`) uses the following calls to look up the types that represent the two mountpoints:

```
s_system_mp_type = secpol_get_type_id(NULL, "system_fs_t");
s_user_mp_type = secpol_get_type_id(NULL, "user_fs_t");
```

The following calls provide the filesystem resource manager with the permission handles for *secpol_check_permission()* (same as the first example):

```
s_mount_perm = secpol_get_permission(NULL, "fs_operation", "mount", 0);
```

The resource manager can check permissions for a request to mount the system mountpoint using the following code:

```
if ( secpol_check_permission(&ctp->info, s_system_mp_type, s_mount_perm) != 0 ) {
    return EPERM;
}
```

Similarly, it can check permissions for a request to mount the user mountpoint using the following code:

```
if ( secpol_check_permission(&ctp->info, s_user_mp_type, s_mount_perm) != 0 ) {
    return EPERM;
}
```

# Definitions in *secpol.h*

*Preprocessor macro definitions for the **secpol.h** header file in the **libsecpol** library*

## Definitions:

```
#include <secpol/secpol.h>
```

**#define SECPOL_INVALID_TYPE 0xffffffff**

**#define SECPOL_DEFAULT_POLICY_FILE "/proc/boot/secpol.bin"**

Default security policy file.

## Library:

**libsecpol**

# secpol_check_permission()

*Test if a given type has a permission*

## Synopsis:

```
#include <secpol/secpol.h>

int secpol_check_permission(const struct _msg_info *info,
                            uint32_t otype,
                            const secpol_permission_t *permission)
```

## Arguments:

***info***

The message information associated with the message that this check relates to.

***otype***

The type of the object being accessed or acted on.

***permission***

A handle to a permission returned by *secpol_get_permission()*.

## Library:

**libsecpol**

## Description:

When a call to *secpol_check_permission()* fails, it indicates that the caller is denied the permission check. The possible errors are:

- EPERM - Permission denied.
- EINVAL - Invalid parameters to function.
- ENOMEM - Out of memory.

In most cases, all errors should be treated identically.

The *otype* argument allows you to restrict the test for a permission to a specific object that the permission accesses or acts on. For more information, see *Customizing permissions using a security policy*.

## Returns:

`0` if the type `ptype` has the indicated permission for the type *otype*, or `-1` if the call failed (`errno` is set).

# secpol_close()

*Close the security policy file*

## Synopsis:

```
#include <secpol/secpol.h>

void secpol_close(secpol_file_t *handle)
```

## Arguments:

***handle***

Handle to the security policy file.

## Library:

**libsecpol**

# secpol_file_t

*Handle for security policy files*

## Synopsis:

```
#include <secpol/secpol.h>

typedef struct secpol_file_s secpol_file_t;
```

## Library:

**libsecpol**

# secpol_flags_e

*Flags for secpol_transition_type(), secpol_posix_spawnattr_settypeid() and secpol_resolve_name()*

## Synopsis:

```
#include <secpol/secpol.h>

enum secpol_flags_e {
    SECPOL_TYPE_NAME = 1
};
```

## Data:

**SECPOL_TYPE_NAME**

The specified name refers directly to a type name.

## Library:

**libsecpol**

# secpol_get_permission()

*Returns a handle to a permission associated with a custom class*

## Synopsis:

```
#include <secpol/secpol.h>

secpol_permission_t* secpol_get_permission(secpol_file_t *handle,
                                           const char *class,
                                           const char *permission,
                                           unsigned flags)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***class***

Name of the class associated with the permission.

***permission***

Name of the permission.

***flags***

Zero or more flags taken from the `secpol_get_permission_flags_e` enumeration.

## Library:

**libsecpol**

## Description:

By default, the *secpol_get_permission()* function will succeed even when no security policy is in use or if the class or permission cannot be found. However, the permission will always be denied. This behavior may be changed by passing appropriate flags.

The possible errors are:

- ENOTSUP - No security policy is in use.
- ENOSYS - The class or permission is not in the policy file or the policy ID is wrong.
- EINVAL - Parameters are not valid for the specified policy file.
- ENOMEM - Out of memory.
- ENOENT - Unable to open the policy file.

**Returns:**

A handle to the permission, or NULL if the call failed (errno is set).

# secpol_get_permission_flags_e

*Flags for secpol_get_permission()*

## Synopsis:

```
#include <secpol/secpol.h>

enum secpol_get_permission_flags_e {
    SECPOL_PERM_STRICT = 1
};
```

## Data:

**SECPOL_PERM_STRICT**

If set, *secpol_get_permission()* will fail and return NULL if there is no policy in use or the specified class or permission does not exist.

If not set, the permission will be allowed for `root` users when there is no policy and never allowed when there is a policy.

## Library:

**libsecpol**

# secpol_get_type_id()

*Return the type ID associated with a type name*

## Synopsis:

```
#include <secpol/secpol.h>

uint32_t secpol_get_type_id(secpol_file_t *handle,
                            const char *type_name)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***type_name***

Name of the type.

## Library:

**libsecpol**

## Returns:

Type ID associated with the name, or SECPOL_INVALID_TYPE if the name was not found.

# secpol_get_type_name()

*Return the type name associated with a type ID*

## Synopsis:

```
#include <secpol/secpol.h>

const char* secpol_get_type_name(secpol_file_t *handle,
                                 uint32_t type_id)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***type_id***

ID of the type.

## Library:

**libsecpol**

## Returns:

Name of the type, or NULL if the type ID was not found.

# secpol_open()

*Open a security policy file*

## Synopsis:

```
#include <secpol/secpol.h>

secpol_file_t* secpol_open(const char *path,
                           uint32_t flags)
```

## Arguments:

***path***

Path to the file. If NULL, the default path is used.

***flags***

Zero or more flags from secpol_open_flags_e ORred together.

## Library:

**libsecpol**

## Description:

In most cases, you set *path* to NULL and *flags* to SECPOL_USE_AS_DEFAULT to ensure a common security policy file is used by all processes and avoid the file being opened multiple times.

However, if you are calling a single function, instead of preceding the call with *secpol_open()* to set the default security policy file, pass NULL for the function's handle. This opens the system's default security policy file for the duration of the function and then closes it.

If SECPOL_USE_AS_DEFAULT is used and a previous call installed a default handle, the call will fail with an errno of EBUSY. This error can usually be ignored as there is already a policy file accessible for other functions to use.

## Returns:

A handle to the file, or NULL if open failed (errno is set).

# secpol_open_flags_e

*Flags for secpol_open()*

## Synopsis:

```
#include <secpol/secpol.h>

enum secpol_open_flags_e {
    SECPOL_USE_AS_DEFAULT = 1
};
```

## Data:

**SECPOL_USE_AS_DEFAULT**

Use handle as default when NULL is specified for the handle in other calls.

## Library:

**libsecpol**

# secpol_permission_t

*Handle for security policy permissions*

## Synopsis:

```
#include <secpol/secpol.h>

typedef struct secpol_permission_s secpol_permission_t;
```

## Library:

**libsecpol**

# secpol_posix_spawnattr_settypeid()

*Update a spawn attribute object to spawn a child with a different type*

## Synopsis:

```
#include <secpol/secpol.h>

int secpol_posix_spawnattr_settypeid(secpol_file_t *handle,
                                     posix_spawnattr_t *attrp,
                                     const char *name,
                                     uint32_t flags)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***attrp***

A pointer to the spawn attributes object to update.

***name***

Name of type, or name to derive type from based on the current type. If a derived type is used, NULL may be passed to use the default name "child".

***flags***

Zero or more flags from `secpol_flags_e` ORred together.

## Library:

**libsecpol**

## Description:

If a security policy is in force, the *secpol_posix_spawnattr_settypeid()* function updates the POSIX spawn attribute structure with the type to spawn as. If a policy is not in force, the function does nothing.

By default, the type ID is selected based on the process' current type and the name passed in (i.e., a derived type). The *name* parameter can also represent the actual type name by passing `SECPOL_TYPE_NAME` in the flags.

If the process spawns multiple child processes that are expected to have different security needs, a different name can be used for each to allow the security policy to fit the security requirements more closely. If the security of all children is likely to be substantially the same, NULL can be passed for the name to use the default name "child". This option can only be used only if a derived type is used.

**Example**

A security policy has the following rules:

```
derive_type resmgr1_t low_priv resmgr1_low_t;
derive_type resmgr1_t high_priv resmgr1_high_t;
```

These rules allow you use the following function call to spawn child processes with a lower level of privilege (of type resmgr1_low_t):

```
secpol_posix_spawnattr_settypeid(NULL, &attr, "low_priv", 0);
```

The following function spawns child processes with a higher level of privilege (type resmgr1_high_t):

```
secpol_posix_spawnattr_settypeid(NULL, &attr, "high_priv", 0);
```

## Returns:

0 if spawn attributes were successfully updated or no update was wanted, or −1 if attribute update failed or no valid type was found.

# secpol_resolve_name()

*Resolve a type name to a type ID*

## Synopsis:

```
#include <secpol/secpol.h>

uint32_t secpol_resolve_name(secpol_file_t *handle,
                             const char *name,
                             uint32_t flags)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***name***

Type name to look up.

***flags***

Zero or more flags from `secpol_flags_e` ORred together.

## Library:

**libsecpol**

## Description:

By default, the type name is assumed to refer to a derived type; that is, a type that is selected based on the process' current type and the name passed in. The *name* parameter can also represent the actual type name by passing `SECPOL_TYPE_NAME` in the flags.

## Returns:

The type ID that is associated with *name* based on the current type of the process, or `SECPOL_INVALID_TYPE` if there is no type ID to return.

# secpol_transition_type()

*Transition to a new type*

## Synopsis:

```
#include <secpol/secpol.h>

int secpol_transition_type(secpol_file_t *handle,
                           const char *name,
                           uint32_t flags)
```

## Arguments:

***handle***

Handle to the security policy file. Usually NULL, which specifies that the default security policy file is used (either the system default or one set using *secpol_open()*).

***name***

Name of type, or name to derive type from based on the current type. If a derived type is used, NULL may be passed to use the default name "run".

***flags***

Zero or more flags from `secpol_flags_e` ORred together.

## Library:

**libsecpol**

## Description:

The *secpol_transition_type()* function attempts to switch to a new type that is either specified as a type name, or, more commonly, as a string (e.g., "run") that is used to derive a type from the current type of the process.

If the function is successful, it indicates that there is a security policy in use, the process is now running with whatever capabilities were deemed appropriate, and that the process should not itself attempt to modify abilities or switch UID as a means of dropping privilege.

If the function fails, the process should instead perform whatever privilege dropping procedure it supports, which in some cases means doing nothing. If the function fails and a policy is in force, system trace events are emitted to allow the problem to be diagnosed.

Behavior should usually be based only on success or failure, not on the specific cause of failure. Failure of this function should not be considered fatal as it will render the program unable to run without security policies.

By default, a derived type is used; that is, the type ID is selected based on the process' current type and the name passed in. The *name* parameter can also represent the actual type name by passing SECPOL_TYPE_NAME in the flags.

It is expected this function will be called only once following initialization. Multiple calls to the function are unlikely to yield any security benefits.

**Example**

A security policy has the following rules:

```
derive_type resmgr1_t run resmgr1_run_t;
derive_type resmgr2_t run resmgr_post_init_t;
```

If a resource manager is started using the security type resmgr1_t and then calls the following function, it switches its type to resmgr1_run_t:

```
secpol_transition_type(NULL, NULL, 0);
```

Alternatively, if the resource manager is started using the type resmgr2_t, this call switches it to resmgr_post_init_t.

## Returns:

0 if the type was successfully switched or −1 if no type change was performed.

# secpol_type_id()

*Return the type ID of the process*

## Synopsis:

```
#include <secpol/secpol.h>

uint32_t secpol_type_id(void)
```

## Library:

**libsecpol**

## Description:

The type ID that the *secpol_type_id()* function returns is accurate only if the process does not call `procmgr_set_type_id` and instead makes any type changes by calling `secpol_transition_type`.

## Returns:

Type ID of the process.

# Index

## T



## U