RED HAT®
ENTERPRISE LINUX®
DEVELOPER PROGRAM

redhat.

WHITEPAPER

SELinux FOR RED HAT DEVELOPERS

HOW TO USE SELinux POLICIES TO ENHANCE DATACENTER SECURITY

EXECUTIVE SUMMARY

This paper discusses how application developers can use SELinux to strengthen datacenter security. While system administrators can take many steps to secure systems, developers can contribute by providing appropriate SELinux policies as part of the RPM (RPM Package Manager) containing their application installation.

TABLE OF CONTENTS

THE NEED FOR SECURITY

Mainstream news reports of serious attacks on corporate infrastructure often underestimate the frequency and scope of breaches, with many organizations unwilling to expose the extent of the data loss. Executives are concerned, as attacks pose a significant threat to valuable business information as well as legal and publicity issues that can affect customer confidence and retention. Companies cannot afford to be complacent or avoid using proper security features due to cost or perceived performance trade-offs. Security at all levels is required, from developers minimizing the potential for risk in their applications, to system administrators limiting, logging, and monitoring access and keeping systems up to date.

Typically, developers do not see themselves as responsible for system security. When an application is implicated in a security breech, customers question vendors and the fallout is hard to ignore. In fact, security failures can create enough upset that customers consider switching to competing applications.

System administrators continue to play a key role in properly protecting individual systems and enterprise-wide infrastructure. Developers need to understand the challenges system administrators face and provide adequate application support so that keeping deployments secure is feasible, reliable, and consistent.

Today, IT organizations use a variety of methods to secure enterprise infrastructure, such as sandboxing, isolating environments with virtualization, deploying firewalls, and performing deep packet inspection. While important, they are not the only techniques that can help keep systems secure. Security-Enhanced Linux (SELinux) provides integrated operating system security enhancements at the kernel-level, and associated tools make it easier to set and enforce security policy decisions.

A LOOK BACK AT THE ROOTS OF SELinux

As early as the 1960s, security-conscious government agencies were concerned with ensuring security at multiple levels of the hierarchy and infrastructure. Various systems were developed, with many concepts still germane today.

- Multi-level security (MLS). In MLS environments, different items can have a different security level. Access can be restricted on a fine-grained basis, with users limited in what they can access and the actions they can perform, reducing the risk posed by unauthorized use.
- Discretionary access controls (DAC). Developers are inherently aware of DAC. In this standard Linux model, permissions set by `chmod` and `chown` identify what can be shared and with whom. Unfortunately, DAC does not identify enough security levels. As a result subverting systems is easy, with users often gaining access to far more information and capabilities than needed. In fact, privilege escalation is common, with users unnecessarily granted root privileges that open up the entire system.
- Mandatory access controls (MAC). In a MAC system, users cannot grant permission (or gain permission) that is not within their defined scope. Depending on the implementation, even root might not be able to perform tasks that have been proscribed by the system architect. History suggests that MAC systems are

used by a small fraction of datacenters and are perceived as difficult to use in production settings.

Many legacy systems, such as Multics, Trusted Solaris, and AIX with Trusted Extensions, provide various forms of MAC. Typically, these systems relied on the role-based access control (RBAC) model. In this model, specific permissions are attached to roles rather than users, and even the superuser is prevented from having arbitrary powers. Users log in (providing an audit trail) and assume a role in order to perform a task. Role granularity can be broad (providing ample avenues for attackers to gain traction) or fine-grained (trading ease-of-use for greater access control). While powerful, RBAC is seldom used effectively in commercial environments.

In 2000, the National Security Agency (NSA) released SELinux, a mechanism and set of applications that provides very fine-grained MAC support in the Linux kernel, as well as deployment tools. While RBAC can be implemented using SELinux kernel facilities, the foundation of the SELinux model controls individual program access rather than specifying user roles. This is a key paradigm shift, one in which developers have a key role to play. Developers should provide default policies for applications that run right out of the box while taking care to minimize the potential for application hijacking. In addition, appropriate support must be in place to adjust system policies, for example, through the use of booleans, file labels, and port definitions, so that system administrators can change installation parameters (such as file locations) and installation scripts.

Since the release of SELinux by the NSA, Red Hat has been at the forefront of making this security technology mainstream, developing tools and sample policies, and contributing them to the Linux community. Red Hat also incorporates the technology into the commercially supported Red Hat Enterprise Linux operating system. This document focuses on SELinux in Red Hat Enterprise Linux 6 environments.

Working with SELinux

In the beginning, when SELinux was configured to actively constrain programs, most applications experienced run-time failures. What was missing were hand-crafted security policies. Today, system administrators still find it necessary to create appropriate security policies, as few applications come with defined policy. As a result, each installation crafts security policies it believes matches application needs, with policy updates often required for every new software release. When done incorrectly, resulting security policies are too permissive (allowing exploitation), overly strict, or simply wrong and can result in unnecessary application failures.

Application developers are in the best position to reliably assert the intent of the application and to establish default security policy. To help this effort, SELinux includes a rich set of fundamental protections. Red Hat provides many infrastructure improvements, enabling application developers to provide appropriate default policies and common customization scripts. When applications provide greater insight, system administrators can focus their efforts on securing the underlying infrastructure, rather than spending time analyzing program behavior and attempting to confine any potential misbehavior of a compromised program according to security requirements.

ARCHITECTURAL CONSIDERATIONS THAT IMPACT SECURITY

Limit where applications perform I/O. If an application must perform I/O to system space, use helper applications to perform those tasks. Restricting I/O to helper applications enables the core application to run locked down, keeping inadvertent errors from compromising the system or providing an attack vector.

Do not create files and directories prematurely. If locations are dynamic, false positives can result if the application runs on a permissively configured system. In locked-down environments, this can cause application failures.

Be mindful of execution. Applications that write code to files and read it in for execution tend to need open access to systems. In addition, applications should ensure input is valid and of appropriate size, rather than interpreting information.

Bias security policies toward security by default. Provide administrative options to weaken security only when needed as a common best practice.

Make policies part of RPM and configuration scripts. This keeps system administrators from having to become experts at application internals or infer the privileges required. Provide documentation for existing applications to help security experts design confinement scenarios, and use tools such as `sepolgen` and `audit2allow` to create draft policies. Ensure that testing and quality assurance (QA) tasks verify policy function and accommodation of reasonable workloads.

Adopt a least-privilege principle. Minimizing the number of openings (ports, I/O channels) and ensuring that every level of the system can only access the information and resources needed for operation is key to closing doors to attackers.

Document security-related considerations. Providing security experts with detailed information, such as identifying spawned helper applications and the permissions they need to operate, enables systems to be configured with greater confidence than when default security policies are not used.

Provide security-related support. Removing barriers for administrators to obtain timely and accurate information quickly is critical to keeping systems secure.

ARCHITECTURAL CONSIDERATIONS

In general, application development tends to focus on performance over security. While such trade-offs made sense years ago when deployments were smaller and had little or no network connectivity, times have changed. It was not uncommon for applications to run as a single process or have large blocks of code, with network and disk I/O interleaved with computation. While some designs used data abstraction and hiding, the use of inlining and inter-procedural analysis by optimizing compilers resulted in what is effectively a high-performance, monolithic process that might access any part of the system based on user workflow. Furthermore, few systems live in isolation. As a result, other systems with which they have a trust relationship must be tightly secured, or at least configured to contain any compromise. Failure to do so exposes the entire datacenter.

While still important, performance must not be the sole consideration for the fundamental architecture of an application. Why? When security is considered only after architecture and implementation, security tends to be poor. Applications often end up running in a sandbox or are retrofitted with security, resulting in additional management complexity. With most enterprise applications running in networked environments, security considerations must become an integral part of software architecture. Fundamental architecture choices dramatically impact the resulting security of an application, and by extension the infrastructure on which it runs.

NUTS AND BOLTS OF SELinux

For readers new to SELinux, this section explains the jargon, application programming interfaces (APIs), and fundamental mindset of SELinux. Readers interested in the theoretical basis of SELinux and the design rationale of the kernel security features can read *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments*[1].

When working with SELinux, the most critical priority is to keep in mind the principle of *least privilege.* Least privilege is the notion that programs should operate with the minimum permissions needed. If a program has a diverse set of permission requirements, two primary approaches are available:

- Divide the program into smaller cooperating processes, with each process having its own least-permissive policy
- Use advanced SELinux capabilities

In reality, most developers are faced with an existing code base and do not have the luxury to re-architect applications around security considerations. While not ideal, incremental security improvements are feasible. For existing applications, developers can generate the basis for a default policy and include it in the RPM. In subsequent releases, developers can adjust the code and default policy to tighten up permissions. Should the need arise, both approaches can be used in the same program. Information on building

1 See http://www.nsa.gov/research/_files/selinux/papers/inevit-abs.shtml.

RPMs can be found at https://access.redhat.com/knowledge/techbriefs/how-build-rpm.

SELinux BASICS

At this point, a complete and validated set of policy files is ready. These policy files should be incorporated into the application RPM and included in the project's source code control system. Many developers may never need to go much further.

To understand in detail what is taking place, it can help to look at SELinux from the bottom up.[2] SELinux completely separates policy from enforcement. While policies rest in the hands of developers and system administrators, enforcement is the kernel's responsibility. At a high level, SELinux is all about labels. Every process, file, directory, and device on a SELinux system has a label. Objects only are allowed to access other objects if the SELinux policy allows objects with their labels to interact. For other object interaction is blocked by the kernel. This is why ensuring that every object has the appropriate label is critical to correct application behavior in the field.

Problems reported by SELinux reports are due to one of the following:

- Labeling errors by developers or administrators
- A confined process is configured differently than the SELinux policy asserts
- A bug in the policy or application
- The system is compromised

Since there are many different kinds of objects, let's discuss a few that application developers are most likely to find critical.

Users

SELinux users are not the same as conventional Linux users. For example, consider a user `dwalsh` with the SELinux user name `staff_u`. This definition enables `dwalsh` to interact with any objects that are permitted to interact with `staff_u`.

It should be noted that SELinux users cannot transition during a user session. (This is unlike the conventional command line, where `su foo` changes the current user to `foo` and `sudo` enables a user to effectively become root.) While the `su` command still "works," the SELinux user does not change. No matter what identity users assume, they do not gain additional access. So, for example, if `dwalsh:staff_u` and `khb:guest` are two users on the system, and `khb` performs `su dwalsh, then khb's` SELinux user name would remain `guest` and `khb` would be unable to manage or interact with facilities not available to `guest`.

2 These fundamentals are adapted from http://selinuxproject.org/page/BasicConcepts and modified by implementation choices in Red Hat Enterprise Linux 6.

The discussion of SELinux users is continued, in the context of *confined users*, later in this document. Confined users help ensure the administrator of an application does not also gain permission to tinker with the rest of the production system, but is limited to only those functions necessary for managing the application.

Roles

SELinux users have one or more roles. What a role means is defined by the system policy. Typical roles include unprivileged user, web administrator, and database administrator.In Red Hat Enterprise Linux, these users are mapped to `staff_r` by default.

Types

An SELinux type is a way of grouping items based on their similarity from a security perspective. The type of a process is known as its domain. By convention, these objects have the `_t` suffix appended to their name. In short, types are the primary means for determining access.

Type enforcement file

Each policy module must have a unique name on the system. As a result, editing an existing policy module must be done carefully, as edits overwrite the existing policy and remove its rules. Improperly editing a policy module for a fundamental system service can result in a broken system.

A type enforcement file (.te) is the source code used to control access. It uses a mini-language based on the traditional `m4` macro language. A `policy_module` macro is available (and recommended) for authoring policy modules. The macro automatically brings in the specific system class and associated permissions. If this macro is not used, a `gen_require` block must be defined manually.

Within a `.te` file, all types must be declared or inherited from a `gen_require` block. This is analogous to `type` declarations in a C program, where many system definitions are inherited via `#include` files. For example, some of the declarations for `myrwhod` include:

```
type myrwhod_t;
type myrwhod_exec_t;
init_daemon_domain(myrwhod_t, myrwhod_exec_t)
permissive myrwhod_t;
type myrwhod_initrc_exec_t;
init_script_file(myrwhod_initrc_exec_t)
type myrwhod_spool_t;
files_type(myrwhod_spool_t)
```

Contexts

All processes and objects have a *context*, known more commonly as a *label*. This attribute is used to determine whether a specific type of access should be allowed between a given process and object. Every SELinux context has three required fields and one optional field: `user:role:type<:MLS range>`. If the user, role, and type fields do not match between the process and object, access is denied (in enforcing mode) or logged (in permissive mode).

Object classes

SELinux defines a number of classes for objects, facilitating grouping permissions by specific classes. Examples include classes for file system and network operations.

Categories of objects, such as `dir` for directories and `file` for files, are default object classes provided by SELinux. Each object class has a set of permissions that define the available ways to access the objects. For example, the `file` object has create, read, write, and delete (unlink) permissions associated with it. See http://selinuxproject.org/page/ObjectClassesPerms for a complete list of object classes and their permissions.

Classes separate different target objects. The most common classes are:

- `file`
- `dir`
- `sock_file`
- `tcp_socket`
- `process`
- `capability`
- `permissions`

Permissions differ per class. Errors are generated if there is an attempt to add a permission that is not defined for the class. Macro definitions are available for higher-level concepts and should be employed whenever possible. This simplifies policy generation and maintenance.

```
file  { read write append ... }
process { fork signal sigkill ...}
capability { setuid setgid ... }
Macro definitions
Many permissions are typically required for one domain to read a file.
read_file_perms, manage_sock_file_perms;
```

Table 1 provides examples.

Table 1: Examples

COMMON FILE PATTERNS
`read_files_pattern(httpd_t, etc_t, net_conf_t)` `/usr/share/selinux/devel/include/support/obj_perm_sets.spt`
MACRO EXAMPLES
`define('read_inherited_file_perms', '{ getattr read ioctl lock}'}` `define('read_file_perms','{ open read_inherited_file_perms }')`

When writing policy by hand, there is a temptation to add permissions one at a time. This is suboptimal, in that policy violations tend to be code-path dependent. Writing a policy change, recompiling the policy, and executing a test suite can be a time-consuming process. Using the provided policy macros is almost always the better approach.

Generally speaking, policies should be generated by developers and tested to ensure end users do not experience avoidable application failures. While it is relatively straightforward to pick appropriate policy macros when one knows the application's intended use cases, reverse engineering intent from observed behavior can prove difficult.

Labels

Types of labels include:

- Object classes. These include processes, files ystem, directories, network ports, devices socket, FIFO, capabilities, and so on. All objects in the operating system are labeled.
- Security contexts. These are labels that contain SELinux security information.

SELinux policies

Labeling policy describes how objects are to be labeled. Access policy describes how subjects access objects. Once defined, these policies are compiled into a binary form and loaded into the kernel, which enforces the policies. A database stores the rules defined by the SELinux policies in effect. These rules define how a process in one context is permitted to operate on an object in another context.

Policy rules

Since the primary security mechanism in SELinux is type enforcement, rules are specified using types for processes and objects.

- **`dontaudit` rules.** These rules should be used when an application needs to go down a different code path. For example, the `pam` libraries default to trying to read the `/etc/shadow` file directly. If denied, the libraries use a helper application. As a result, the following must be added to allow the code to use the alternate path without generating active vector cache (AVC)messages: `dontaudit` *`DOMAIN`* `shadow_t:file read;` Whenever possible, developers should employ helper applications. This simplifies the policy-writing task and makes application intent more transparent.

- **`allow` rules.** When editing an existing (or cloned) `.te` file, it is common to add `allow` rules. (The default action is to deny everything.) After defining a new type, allow rules must be added or attributes that have associated allow rules must be added, or the type will not be able to do anything.
- **`neverallow` rules.** These rules should only be used by distributions. As a result, they are not discussed in this document.
- **`auditallow` rules.** Seldom used, these rules are only for loading a policy or changing booleans in the distribution policy. While these rules can be used for auditing, the Linux audit subsystem is a superior solution.

The following policy example allows `user_t` labeled objects to access their home directories with full permissions, followed by a rule that allows `user_t` only to read files irrespective of the file's read/write permissions.

```
allow user_t user_home_t:file {create read write unlink}
allow user_t user_home_t:file { read }
```

Interfaces (policy function calls)

Interfaces are the methods or functions used to interact with module types. When writing a policy module, avoid using the type of another module. Always use the module's interfaces. When defining types within a module, a best practice is to define multiple interfaces to allow other domains to interact with the module's types.

Each policy module should have an interface called *`DOMAIN`*`_admin` that allows a confined administrator to completely administer the domain.

Interfaces are stored in `.if` files, such as:

- `/usr/share/selinux/devel/include/kernel/files.if`
- `files_type(shadow_t)`
- `init_system_domain(rwhod_t, rwhod_exec_t)`
- `corenet_tcp_connect_mssql_port(httpd_php_t)`
- `apache_admin(webadm_t)`

Attributes

Attributes provide a way to group types. Adding an attribute to a type causes the type to inherit all the `allow` and `dontaudit` rules associated with the attribute. Attributes can be used in the source or target portion of the SELinux rule. For example:

- `attribute file_type`
- `type etc_t, file_type`
- `allow rpm_t file_type:file manage_file_perms;`

- `allow domain self:process fork;`

Interfaces used to assign attributes include:

- `files_type(etc_t)`
- `domain_type(httpd_t)`

Control writing

A typical attack vector involves writing a file that another process reads. As a result, a process that owns data should be assigned its own type. In addition, user-level applications should not be allowed access to system files types, such as:

- `etc_t`
- `usr_t`
- `var_lib_t`
- `var_run_t`
- `root_t`

If the data is owned by another domain, the appropriate interface must be used. For example:

```
http_sys_content_t use apache_write_content(dictd_t)
```

In addition, allowing one process to send signals to another process, such as `sigkill`, can result in denial-of-service (DoS) attacks. There are times when sending signals from one process to another is appropriate, such as:

```
allow guest_t guest_t:process sigkill;
allow guest_t guest_dbusd_t:process sigkill;
```

Capability class

In striving to limit the power of root, SELinux has defined 34 capabilities at the time of this writing. These capabilities are listed in the `/usr/include/linux/capability.h` file. A best practice is to design applications so they are managed by a contained user. See http://www.freetechbooks.com/efiles/selinuxnotebook/The_SELinux_Notebook_The_Foundations_3rd_Edition.pdf for more information.

Interface files

All of the policy rules that describe how objects are labeled and are allowed to transition and interact are encoded as *interface files*. These files act as the source code to the policy compilation process. The following examples should be helpful in making these various concepts concrete.

Example: Making incremental security improvements to an existing application

Consider the case of making incremental security improvements to an existing application throughout the release cycle. For many applications, the use of Red Hat SELinux tools (CLI and GUI) provides the bulk of policy generation required. Developers should try these tools and become comfortable with the infrastructure before applying the framework to their application. Organizationally, it can be helpful to set up a working group that includes test engineers, which allows security policies to be wired into integration and testing plans to be implemented as policies.

Fortunately, many applications only need to read from--and perform I/O to--a few directories and use a handful network ports. Ideally, read-only data would be restricted to one directory with read-write data residing in a separate directory. Similarly, if there are only a few network ports to permit, restricting all access to other directories and ports is easy, resulting in a secure application with minimal design effort. Since Linux developers put a great deal of effort into making startup daemons and many of the most basic system tools are SELinux-policy friendly, one of these can serve as an example.

Preparation

Before starting, prepare a special test system or a virtual system that can be discarded after these experiments. The first step is to add the security tools packages:

```
# yum install policycoreutils-gui selinux-policy-targeted
```

Let's take a look at a modified `rwho`[3] daemon (`myrwhod`[4]). It is similar to the `who` command, but is for users logged into hosts on the local network. The following sections examine the steps necessary to create, compile, and install the necessary SELinux security policy. It generates the following files:

- Type enforcement file (.te). This file contains all of the code to implement the SELinux policy used to confine the application under development.
- File context file (.fc). This file contains the mappings between files and file contexts.
- Interface file (.if). This file contains all of the interfaces other domains might want to use to communicate with the domain and file types created by the application under development.
- Shell script (.sh). This script is used to compile, install, and fix the labeling on the system under test.

3 See http://www.linuxcommand.org/man_pages/rwho1.html
4 See http://people.fedoraproject.org/~dwalsh/myrwhod/

Step 1: Download and install the myrwhod RPM[5].

```
$ wget http://people.fedoraproject.org/~dwalsh/myrwhod/myrwho-0.17-34.el6.i686.rpm
```

The next set of steps requires superuser status. Rather than constantly typing `sudo`, attain superuser privileges.

```
$ su
```

Install the demonstration `myrwhod` package.

```
# rpm -i myrwho-0.17-34.el6.i686.rpm
```

Verify the installation.

```
# rpm -q myrwho
myrwho-0.17-34.el6.i686
```

5 We start with the CLI tools, as they are the most natural approach for most developers with an existing application. The confined users discussion later in this document uses the GUI tools, as the more elaborate choices available are well suited to GUI prompts. Developers can choose which style of tool to use. Note that the source RPM requires additional infrastructure to install via `rpm -i`. For just viewing the code, the easiest approach is `rpm2cpio | cpio -idmv`.

Step 2: Prepare the system to create a policy module.

Make sure the `myrwhod` service is installed and functional.

```
# service myrwhod start
# service myrwhod status
    myrwhod (pid 6883) is running...
# myrwho localhost
# service myrwhod stop
```

Create a clean subdirectory for policy generation.

```
# mkdir /root/myrwhod
# cd /root/myrwhod
```

Step 3: Create the initial policy module draft.

The `sepolgen` command-line tool is used to create a policy template from an existing executable . Using `rwhod` as an example:

```
# sepolgen /usr/sbin/rwhod
    Generating Policy for /usr/sbin/myrwhod named myrwhod
    Loaded plugins: auto-update-debuginfo, presto, refresh-packagekit
    Created the following files:/
    ./myrwhod.te # Type Enforcement file
    ./myrwhod.if # Interface file
    ./myrwhod.fc # File Contexts file
    ./myrwhod.sh # Setup Script
```

Step 4: Create the final policy module.

This step starts an iterative process: run the application with `audit2allow`, look for warnings or errors, and add permissions until warnings cease.

[1] Repeat until no warnings:

Install the generated policy. (This is analogous to a compile and installation step.)

```
# sh ./myrwhod.sh
Building and Loading Policy
+ make -f /usr/share/selinux/devel/Makefile
Compiling targeted myrwhod module
/usr/bin/checkmodule:  loading policy configuration from tmp/myrwhod.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing binary representation (version 10) to
tmp/myrwhod.mod
Creating targeted myrwhod.pp policy package
```

```
rm tmp/myrwhod.mod.fc tmp/myrwhod.mod

+ /usr/sbin/semodule -i myrwhod.pp

+ /sbin/restorecon -F -R -v /usr/sbin/myrwhod

/sbin/restorecon reset /usr/sbin/myrwhod context system_u:object_r:bin_t:s0-
>system_u:object_r:myrwhod_exec_t:s0

+ /sbin/restorecon -F -R -v /etc/rc.d/init.d/myrwhod

/sbin/restorecon reset /etc/rc.d/init.d/myrwhod context
system_u:object_r:initrc_exec_t:s0-
>system_u:object_r:myrwhod_initrc_exec_t:s0

+ /sbin/restorecon -F -R -v /var/spool/myrwho

/sbin/restorecon reset /var/spool/myrwho context
system_u:object_r:var_spool_t:s0->system_u:object_r:myrwhod_spool_t:s0

[root@RHEL6 myrwhod]#
```

Run the application with as many varied inputs as required for the supported use cases:

```
# service myrwhod start

# myrwho localhost

# service myrwhod status

# service myrwhod stop
```

The `myrwhod` example is very simple. It uses only a single port, and the code path is invariant with respect to the `myrwho` target system. So these are the only use cases that cover the supported usage (start, lookup, check status, and stop). Most AVC applications are more complicated, and a full test suite might need to be run under `audit2allow`.

View the AVC warning and error output with both the `-la` and `-laR` options. Note that the `-R` option searches through `/usr/share/selinux/devel/include` interface files looking for the best match.

```
# audit2allow -la

#============= myrwhod_t ==============

allow myrwhod_t initrc_var_run_t:file { read lock getattr open };

allow myrwhod_t proc_t:file { read getattr open };

#!!!! This avc can be allowed using the boolean 'allow_ypbind'

allow myrwhod_t rwho_port_t:udp_socket name_bind;

allow myrwhod_t rwho_spool_t:dir search;

#!!!! This avc can be allowed using the boolean 'allow_ypbind'

allow myrwhod_t self:capability net_bind_service;
```

```
# audit2allow -laR
require {
        type myrwhod_t;
        class capability net_bind_service;
}
#============= myrwhod_t ==============
#!!!! This avc can be allowed using the boolean 'allow_ypbind'
allow myrwhod_t self:capability net_bind_service;
corenet_udp_bind_rwho_port(myrwhod_t)
init_read_utmp(myrwhod_t)
kernel_read_system_state(myrwhod_t)
rwho_search_spool(myrwhod_t)
```

Examine the code and see if there is a different approach. For example, if the `audit2allow -la` command returned `allow abc_t xyz_t:file {read getattr lock};` yet the only interface in the code was `xyz_rw_file()` and `audit2allow -laR` returned `xyz_rw_file(abc_t)`, it would be preferable to use the raw rules and develop a new interface, such as `xyz_read_file()`.

Of course, there are more elaborate possibilities. There can be attempts to write to directories which should not be written to, or the order in which things are done might be suboptimal. In all cases, avoiding doing something that triggers an AVC, rather than just allowing it (or suppressing the AVC generation), is the best approach.

Assuming there are no code changes advisable, add the necessary policy changes:

```
# audit2allow -laR >> myrwhod.te
```

By appending the output to the `application.te` file, additional policy permissions are added automatically. These permissions are required to allow the application to run as intended. In the absence of code changes, such additional policy permissions tend to open up the system more than is optimal. Developers should seek to change the application rather than simply disabling the AVC warnings in this fashion.

End repeat loop (return to [1] until AVC messages no longer are produced)

Background about what happens in the above loop can facilitate understanding. The `audit2allow` command searches through the installed policy interface files on disk and attempts to find the best match for the generated AVC messages. These interface files are installed under the `/usr/share/selinux/devel` directory. In this example, the "`kernel_read_system_state(rwho_t)`" message was found. Note that any AVC that requires a domain to interact with itself includes the self qualifier.

The tool did not find an interface to match the generated messages:

```
allow rwho_t initrc_var_run_t:file { read write getattr lock };
```

Following are the AVCs generated in the `/var/log/audit/audit.log` file:

```
type=AVC msg=audit(1184874740.520:1685): avc:  denied  { read write } for
pid=18340 comm="rwhod" name="utmp" dev=dm-0 ino=3178503
scontext=system_u:system_r:rwho_t:s0
tcontext=system_u:object_r:initrc_var_run_t:s0 tclass=file

type=PATH msg=audit(1184874740.520:1685): item=0 name="/var/run/utmp"
inode=3178503 dev=fd:00 mode=0100664 ouid=0 ogid=22 rdev=00:00
obj=system_u:object_r:initrc_var_run_t:s0
```

These messages indicate the `myrwhod` daemon is trying to read and write the `/var/run/utmp` file. The library for interacting with the `utmp` file always attempts to open the file, and read and write it before falling back to read-only mode. Yet `myrwhod` should not be able to write over the `utmp` file, as it could contain important security data. Looking in the `/usr/share/selinux/devel` directories, two calls are found: `init_read_utmp(rwho_t)` and `init_dontaudit_write_utmp(rwho_t)`. The first interface call allows `myrwhod` to read the `utmp` file. The second interface call instructs the kernel to stop generating AVC messages when `myrwhod` attempts to write to the `utmp` file. Changing the code to *not* write to the `utmp` file at all is preferable to disabling AVC messages about failed writes.

Once the policy is working, it can be added to the policy RPM and released to the testing group. The final SELinux policy source files should be checked into the same source code repository as the application, and maintained in parallel as the code evolves. Detailed information on building RPMs can be found at https://access.redhat.com/knowledge/techbriefs/how-build-rpm.

Finally, study the AVC messages to determine if there ought to be a code change rather than a policy change. Often, minimizing the number of permissions is possible with a modest code rewrite. In the example above, adjusting the library to have a read-only call would be ideal, however, a full treatment is a topic for another paper.

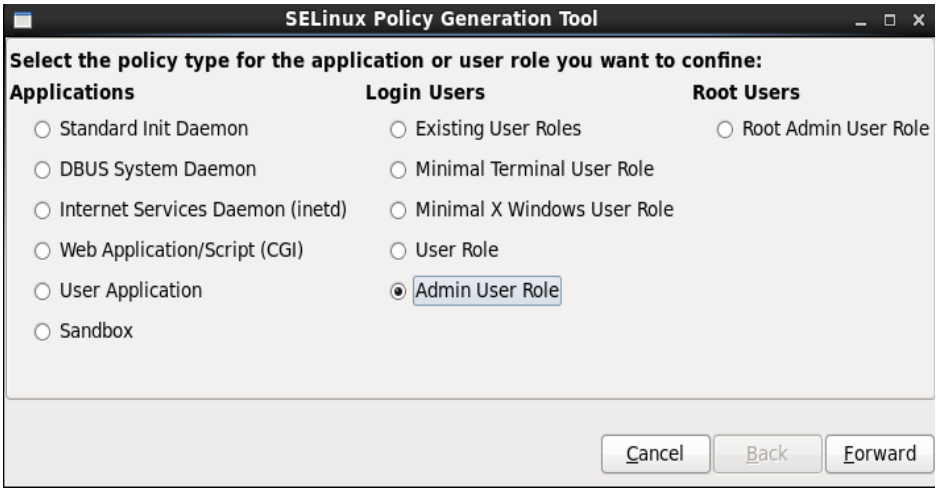
Confined users (RBAC)

SELinux enables users to be confined. By default, a few SELinux user roles are defined by the system (normal, root, X Window-enabled guest, and CLI-enabled guest). Similar to the classic RBAC model, the span of control for a user can be curtailed. Developers should consider whether their application can be managed by an appropriately confined user, excluding as many system, hardware, and application controls from the application administrator as possible. If so, appropriate classes should be incorporated into the system policy as part of RPM installation.

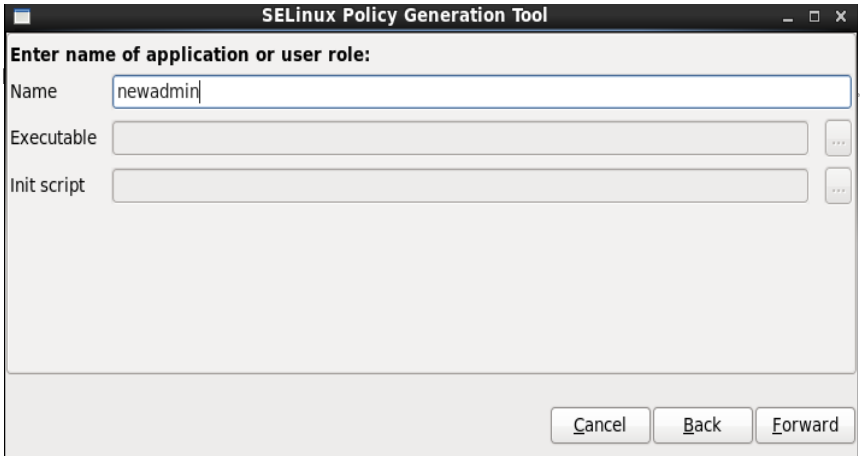
For example, consider the case of a confined user with permission to run an application, alter its configuration files, start and stop the specific services the application stack requires—yet the user is denied permission to write to any other system files or control other system or application services. Accomplishing this requires creating a new user and a bit of policy coding.

For simplicity, assume the application requires `MySQL` and `httpd`. Administrators of this application will need to manage these tools. Of the SELinux user roles provided in Red Hat Enterprise Linux 6, `staff_u` comes closest to providing the privileges required. Let's extend `staff_u` so that all of the existing (and other) users with this SELinux label (and with suitable entries in the `sudoers` file) are able to administer this application stack as well. This is typically not a concern, because the application stack often runs in a virtual machine where only the appropriately authorized users and administrators are `staff_u`. In a system where finer control over users is required, more elaborate policy can be written for an entirely new SELinux user.

Create a new SELinux user using `selinux-polgengui`.

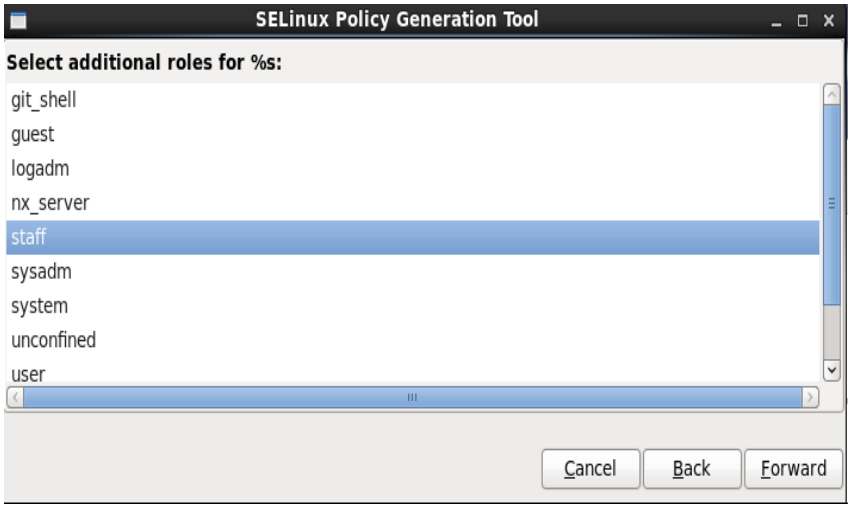


Since an executable or `init` script is not needed to create the new user, those items can remain blank.



The new user must be assigned a *role*. Using the standard roles provided, select `shutdown`. This allows the application's administrator to reboot the system, if necessary.

As shown in the next pane, assign `newadmin` to the existing SELinux role `staff`.



The next several panes deal with network port restrictions. This example grants permission to all ports. In most applications, the confined administrator usually only needs access to a few ports.

Next, generate the policy into a new directory.

CONFINED USER REFERENCES

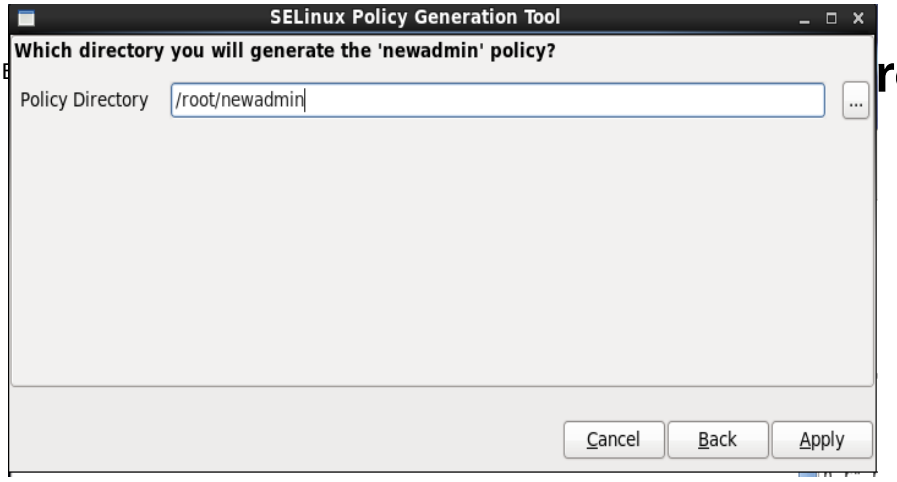
Confining Users with Predefined SELinux Security Policies in Red Hat Enterprise Linux 6

https://access.redhat.com/knowledge/sites/default/files/attachments/confining_users_with_selinux_in_rhel_6_0.pdf

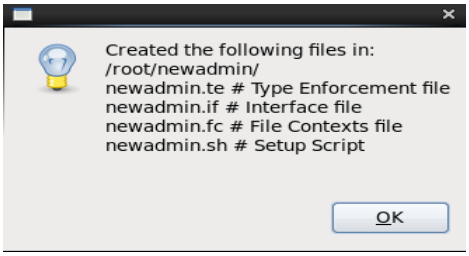
Video presentations that explain the consequences of inappropriate policies:

https://access.redhat.com/knowledge/videos/214723

https://access.redhat.com/knowledge/videos?title=&field_vid_reference_product_tid=All&page=5



This generates the necessary policy source files.



The following shows policy compilation and installation.

```
# sh ./new
# admin.sh
Building and Loading Policy
+ make -f /usr/share/selinux/devel/Makefile
Compiling targeted newadmin module
/usr/bin/checkmodule:  loading policy configuration from tmp/newadmin.tmp
/usr/bin/checkmodule:  policy configuration loaded
/usr/bin/checkmodule:  writing binary representation (version 10) to
tmp/newadmin.mod
Creating targeted newadmin.pp policy package
rm tmp/newadmin.mod.fc tmp/newadmin.mod
+ /usr/sbin/semodule -i newadmin.pp
+ /usr/sbin/semanage user -a -R 'newadmin_r staff_r system_r' newadmin_u
+ '[' '!' -f /etc/selinux/targeted/contexts/users/newadmin_u ']'
+ cat
```

Having created the policy, create the new administration user and prepare it for use.

```
# useradd -Z newadmin_u newadmin
# passwd newadmin
Changing password for user newadmin.
passwd: all authentication tokens updated successfully.
```

The final step is to edit the `sudoers` file to allow `newadmin` to become `newadmin_r:newadmin_t` when run as `root`.

```
# visudoer
newadmin  ALL=(ALL)       ROLE=newadmin_r TYPE=newadmin_t ALL
```

Confining users is a useful technique. Confined users can be used to limit administrative or normal user access to the minimum amount of access a specific user role requires, whether it be access to applications, devices, or ports.

Confined applications

Just as SELinux enables user roles to be locked down, applications can also be confined. The basis for these confinement techniques are SELinux *transitions*—a mechanism that allows labeling rules to specify the way a target type transitions. For example, file transitions consist of statements, such as: “When a process labeled `dictd_t` creates a file or directory in a directory labeled `var_run_t`, the file or directory should be created with the `dictd_var_run_t type`.” This statement is encoded as follows:

```
filetrans_pattern(dictd_t, var_run_t, { file dir }, dictd_var_run_t)
```

When a confined application tries to execute a file with a *`DOMAIN`*`_exec_t` label, execution can occur in two ways. The domain can execute in the current label or transition to another domain. For example, assume `audit2allow` generated the following `allow` rule when `dictd` tried to execute a file labeled `sendmail_exec_t;`

```
allow dictd_t sendmail_exec_t:file { execute read ...
```

The `sendmail` command can run as `dictd_t` using `can_exec(dictd_t, sendmail_exec_t)`. Alternatively, the command can transition to the full `sendmail` domain using `sendmail_domtrans(dictd_t)`, which runs `sendmail` as `sendmail_t;`.

Application developers should strongly consider confining their applications. Most applications should have a fairly small set of SELinux transitions that need to be permitted.

SELinux modules

While policy modules can be written by hand using `checkmodule` and `semodule_link`, a `Makefile` is provided that simplifies the policy module compilation process and reduces the likelihood of error. The `Makefile` is executed using the `make` command:

```
# make -f /usr/share/selinux/devel/Makefile
```

The policy module can be installed using the `semodule -i DOMAIN.pp` command. This command typically takes several seconds to execute, as it compiles and verifies the entire policy (not just changes) and loads the policy module into the system.

Installing policy modules that include file context files results in modification of the default file context on the system. Note that the file context is not applied to the file system. The `restorecon` command must be used to modify the extended attributes on the file system. A complete sequence is shown below.

```
# make -f /usr/share/selinux/devel/Makefile
# semodule -i dictd.pp
# restorecon -R /var/run/dictd.pid
```

The first line installs the policy into the system, and the second line assigns the appropriate file contexts. More information on packaging can be found in the shipping policies for Red Hat Enterprise Linux located at http://danwalsh.livejournal.com/49762.html. Typically, policies must be compiled for each major Red Hat Enterprise Linux release to be supported. Packaging SELinux policies for Fedora distributions are available at https://fedoraproject.org/wiki/SELinux_Policy_Modules_Packaging_Draft.

Workflow

After the initial policy is drafted, compiled, installed, and has labels applied, the project is ready for preliminary quality assurance (QA) testing. A best practice is testing the application using test suites running in permissive mode. During execution, AVC messages are generated and stored in the `/var/log/audit/audit.log` file. An example of this iterative process can be found earlier in this document.

CONCLUSION

Developers should provide default policy source and precompiled files as part of an RPM distribution. The policy should typically include user and application confinement as appropriate.

Including policy source and scripts as part of the RPM allows system administrators to maintain the security of their system with minimal additional manual overhead. Maintaining appropriately limited permissions for each and every application running on a production system is required to keep datacenters secure. Relying on the system administrator to determine what an application's intent is, or what all of the supported use cases are, is an unreasonable burden to place upon system administrators.

FOR MORE INFORMATION

More information on SELinux can be found at:

- Dan Walsh's blog: danwalsh.livejournal.com
- *Red Hat Enterprise Linux 6, Security-Enhanced Linux User Guide,* Edition 3 https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/index.html
- Red Hat videos: http://www.redhat.com/sourcelibrary/videos
- *SELinux by Example*, Prentice Hall http://www.informit.com/store/selinux-by-example-using-security-enhanced-linux-9780131963696
- *The SELinux Notebook*, Second Edition, http://freecomputerbooks.com/The-SELinux-Notebook-The-Foundations.html
- https://access.redhat.com/knowledge/videos?title=&field_vid_reference_product_tid=All&page=5

About the author

Dan Walsh has over 30 years of experience in the computer field. He has spent the majority of his career working on security applications and platforms, including the Athena Project at Digital Equipment Corporation, the AltaVista Firewall and AltaVista Tunnel (VPN) products, HackerShield (a vulnerability assessment solution) and BVControl for UNIX. At Red Hat, Dan has led the SELinux project, concentrating mainly on the application space and policy development. Dan graduated with a B.A. in Mathematics from the College of the Holy Cross and an M.S. in Computer Science from Worcester Polytechnic Institute.

ADDENDUM

SELinux sandboxing

While not part of the default installation, sandboxing[6] is supported by SELinux. System administrators can install the sandbox `policycoreutils-sandbox` package with the Add/Remove Program GUI dialog or via command line:

```
# sudo yum install policycoreutils-sandbox
```

Applications that do not have policies defined, or are difficult to tightly confine, can be executed in the sandbox with the following command:

```
# sandbox $app-name
```

Applications that need permissions beyond simple reading and writing to `stdin` and `stdout`, or those that are handed other file descriptors, can use the `-M` (mount) and `-X` (window system support) options. See the `sandbox(8)`man page for detailed information. For example, the `sandbox -X firefox` command can be used for a web browser that is used locally. It results in a web browser running in a sandboxed window that cannot connect to outside sites. Web access can be allowed using the `sandbox -X -t sandbox_web_t firefox` command.

A recommendation is to develop applications so they do not need to run in a sandbox. If a sandbox is required during initial efforts to make an application secure, development and test organizations should test and document the sandbox openings that are required. Appropriate policies and scripts should be provided to keep users safe by default.

More information on sandboxing can be found at http://danwalsh.livejournal.com/31146.html.

For historical perspective, the sandbox facility was introduced by the author. See http://danwalsh.livejournal.com/28545.html, http://danwalsh.livejournal.com/31146.html, and http://people.fedoraproject.org/~dwalsh/SELinux/Presentations/sandbox.pdf for more information.

6 Sandboxing is the process of taking an application not designed to be secure, and putting it into a *sandbox*. A sandbox is an area with a special limited permission state. Using a sandbox ensures the rest of the system is protected from the activity of the sandboxed object or application. See http://en.wikipedia.org/wiki/Sandbox_(computer_security).

ABOUT RED HAT

Red Hat was founded in 1993 and is headquartered in Raleigh, NC. Today, with more than 60 offices around the world, Red Hat is the largest publicly traded technology company fully committed to open source. That commitment has paid off over time, for us and our customers, proving the value of open source software and establishing a viable business model built around the open source way

SALES AND INQUIRIES

NOTE TO DESIGNER: PLEASE PROVIDE THE CORRECT CONTENT HERE

NORTH AMERICA
1–888–REDHAT1
www.redhat.com

EUROPE, MIDDLE EAST AND AFRICA
00800 7334 2835
europe.redhat.com
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac.redhat.com
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
latam.redhat.com
info-latam@redhat.com