BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EXTENDING AUDIT2ALLOW TO PROVIDE MORE RESTRICTIVE SOLUTIONS

ROZŠÍŘENÍ NÁSTROJE AUDIT2ALLOW PRO POSKYTOVÁNÍ VÍCE OMEZUJÍCÍCH ŘEŠENÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAN ŽÁRSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/20725/2017/xzarsk03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Žárský Jan**
Obor: Informační technologie
Téma: **Rozšíření nástroje audit2allow pro poskytování více omezujících řešení**
Extending audit2allow to Provide More Restrictive Solutions
Kategorie: Operační systémy

Pokyny:

1. Nastudujte technologii SELinux. Nastudujte projekt audit2allow. Seznamte se s existujícími bezpečnostními politikami operačních systémů Fedora a RHEL.
2. Analyzujte současné problémy s méně omezujícími návrhy úprav bezpečnostní politiky poskytované nástrojem audit2allow. Navrhněte rozšíření audit2allow, které bude podporovat více omezující rozšíření bezpečnostní politiky SELinux (např. úprava pouze nekritických částí politiky, úprava politiky na základě hodnot argumentů systémových volání, úprava politiky pouze pro vybraný přístup k souborovému systému).
3. Implementujte vybraná rozšíření bezpečnostních politik v nástroji audit2allow.
4. Ověřte funkcionalitu řešení na základě umělé testovací sady.

Literatura:

- Vermeulen, Sven. Selinux System Administration: Ward Off Traditional Security Permissions and Effectively Secure Your Linuxs Systems with Selinux. second ed. Birmingham, UK: Packt Publishing, 2016.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

The thesis analyzes the role of the audit2allow utility in troubleshooting Security-Enhanced Linux denials and proposes extensions that will provide more restrictive and more secure solutions to the user. In first part, basic concepts of SELinux are explained. The second part contains analysis of situations when audit2allow provides ineffective and insecure solutions. Third part describes implementation of chosen extensions to audit2allow that provide more restrictive and secure solutions. The last part describes testing of these extensions.

Abstrakt

Bakalářská práce rozebírá roli nástroje audit2allow při řešení zamítnutí přístupu systémem Security-Enhanced Linux a navrhuje rozšíření nástroje tak, aby poskytoval více omezující a bezpečnější řešení uživateli. První část popisuje základní koncepty systému SELinux. Druhá část obsahuje analýzu situací, kdy nástroj audit2allow poskytuje řešení, která jsou neefektivní a potenciálně nebezpečná. Třetí část popisuje implementaci vybraných rozšíření, které poskytují více omezující a bezpečnější řešení. Poslední část popisuje testování těchto rozšíření.

Keywords

SELinux, audit2allow, security, mandatory access control, security policy

Klíčová slova

SELinux, audit2allow, bezpečnost, mandatorní řízení přístupu, bezpečnostní politika

Reference

ŽÁRSKÝ, Jan. *Extending audit2allow to Provide More Restrictive Solutions.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

TODO

Extending audit2allow to Provide More Restrictive Solutions

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Aleš Smrčka, Ph.D. The supplementary information was provided by Mgr. Miloš Malík, Petr Lautrbach, Lukáš Vrabec and Ing. Vít Mojžíš. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Jan Žárský
April 22, 2018

Acknowledgements

I would like to thank Mgr. Miloš Malík, Petr Lautrbach, Lukáš Vrabec, and Ing. Vít Mojžíš for valuable advice that helped me write this thesis.

Contents

Chapter 1

Introduction

Security-Enhanced Linux is a mandatory access control mechanism used in Linux distributions. It extends the traditional Unix file permissions using security policies that cannot be overridden by users. The *audit2allow* utility is one of several tools used by system administrators to troubleshoot SELinux denials. SELinux security policy developers use audit2allow to create a basis for security policy modules for their products. The audit2allow utility analyzes SELinux denials and creates snippets of security policy that can be loaded into the security policy to allow the operations that were denied.

In certain situations, the audit2allow utility fails to provide effective and secure solution. There are several reasons for that. Users often try to use audit2allow to solve problems that does not require new rules in policy. Some solutions are not effective because audit2allow does not recognize new types of statements in SELinux policy. The audit2allow utility cannot handle denials caused by mislabeled objects on the system.

Users that are not familiar with SELinux cannot recognize limitations of audit2allow and end up with insecure policy modules on their system. This thesis aims to analyze different causes of SELinux denials and evaluate the quality of solutions provided by the audit2allow utility. Situations that are best resolved using other tools should be detected by audit2allow and user should be warned. Selected improvements to audit2allow were implemented.

Second chapter of the thesis presents Security-Enhanced Linux, introduces SELinux policy language, describes auditing of security events, and describes in detail the audit2allow utility. The third chapter discusses problems that arise with audit2allow usage. The fourth chapter describes implementation details of selected improvements to audit2allow.

Chapter 2

Security-Enhanced Linux and audit2allow

This chapter describes basic concepts of Security-Enhanced Linux, introduces the SELinux security policy language, provides overview of the Linux Audit System, and describes in detail the audit2allow utility.

2.1 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a mandatory access control mechanism that consists of kernel modifications and user-space tools and is a part of several Linux distributions.

2.1.1 Purpose of SELinux

Without SELinux, operating system relies on traditinal access control methods such as file permissions. Users can grant insecure file permissions to others or gain access to files that they do not need [10]:

- Users can reveal sensitive information by setting world readable permissions on their files. For example, they can set read permission for everyone on SSH keys in the `~/.ssh/` directory.
- Processes can change security properties. For example, mail client can make user's mail readable by other users.
- Processes inherit user's rights. For example, every application, even though it may be compromised, is able to read all user's files.

SELinux enforces a security policy that cannot be overriden by users. Application are allowed to perform only actions they need for normal operation, everything else is denied by default. Applications do not need to be aware of SELinux. When an action is denied, it is reported via "access denied" error code to the application [4].

2.1.2 SELinux Components

SELinux is composed of kernel and userspace part [14, pp. 19–22]. Main components of SELinux, are shown in figure 2.1.

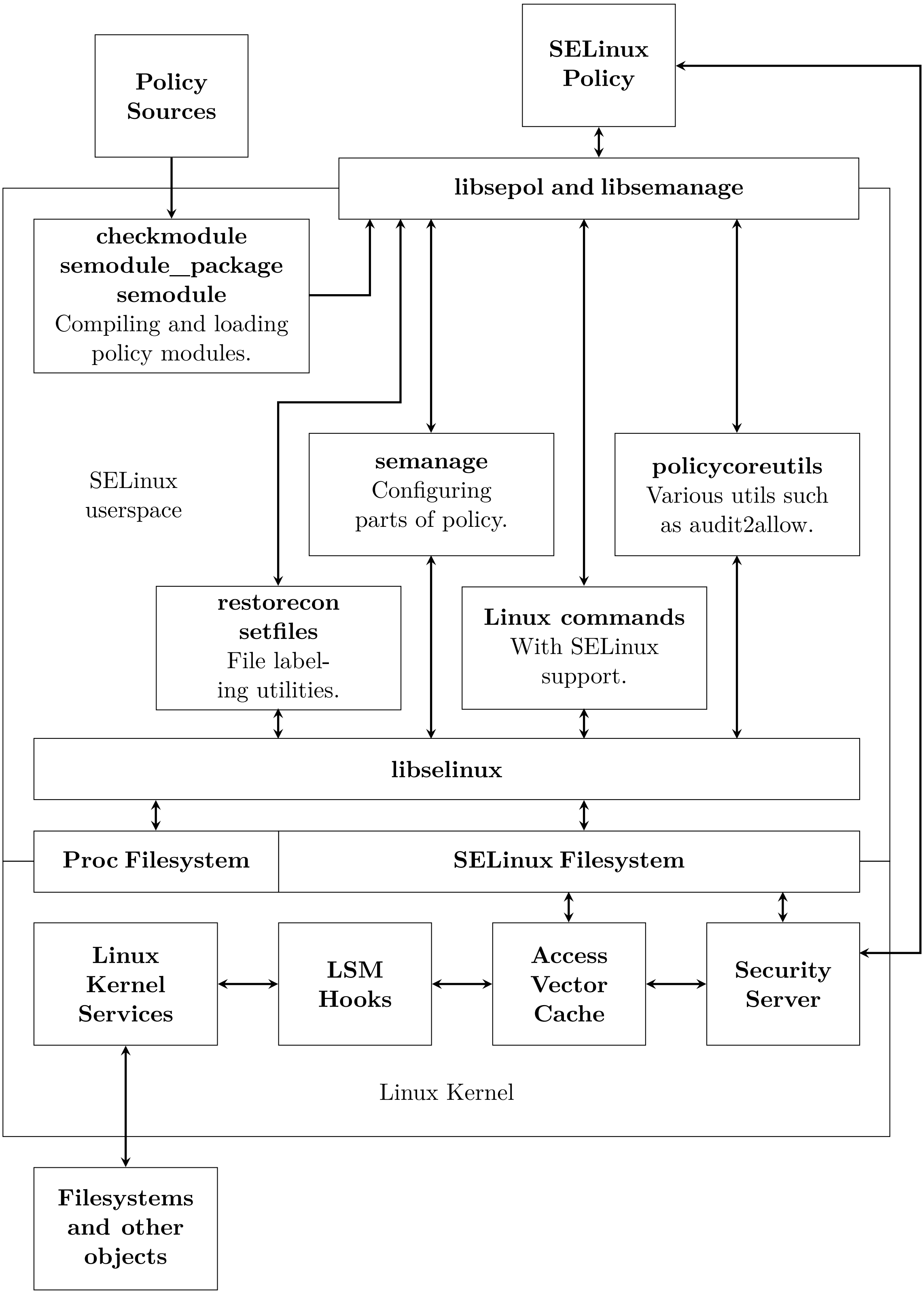

Figure 2.1: Main SELinux components.

libsepol and libsemanage Libraries for working with SELinux binary policy and policy infrastructure. The libsepol library is used for example for loading policy modules into the active security policy. The libsemanage library is used for example for assigning security contexts to TCP or UDP ports.

libselinux API for implementing SELinux-aware applications. For example, SELinux-aware window manager can use the libselinux library to compute security contexts of its objects.

checkmodule, semodule_package, semodule Utilities that compile SELinux policy and load it into the kernel.

semanage Utility for configuring various parts of policy, for example setting contexts for TCP and UDP ports. It uses mainly the libsemanage library.

restorecon and setfiles Utilities for restoring default context of files (see section 2.4).

policycoreutils Various utilities for working with and troubleshooting of SELinux, for example the audit2allow utility described in section 2.7.

Modified Linux Commands Standard Linux commands such as ls or ps, modified to support SELinux.

SELinux and proc filesystem Userspace tools communicate with kernel security server via the `/proc` and `/sys/fs/selinux` filesystem.

Security Server Makes security decisions. It is embedded in the kernel and it obtains the security policy via userspace tools. Security server does not enforce the decision, it only states whether the operation is allowed or not.

Access Vector Cache Caches security decision made by security server.

Linux Security Module Hooks Call SELinux Security Server.

2.2 Basic Concepts

This section defines basic terms related to SELinux, such as mandatory access control, type enforcement, multi-level and multi-category security, security context, and others.

2.2.1 Subjects and Objects

There are two basic entities in SELinux [14, p. 29]:

Subject is an entity that causes information to flow among objects or changes the system state. Within SELinux, a subject is an active process that can access objects. A process can also be an object, for example when a process is sending signal to another process, the process receiving the signal is treated as an object.

Object is a system resource such as file, socket, pipe, TCP or UDP port, network interface, semaphore or shared memory segment.

2.2.2 Mandatory Access Control

SELinux provides mandatory access control mechanism that extends the discretionary access control mechanisms present in Linux kernel.

Discretionary Access Control

Discretionary access control (DAC) is defined by *Trusted Computer System Evaluation Criteria* (TCSEC) standard [11]. System with DAC must enable users to protect their data by controlling access to their data, e.g. by setting permissions for other users or user groups. In DAC, users make security decisions by specifying who can access their data. The disadvantage is that users can propagate sensitive information.

Linux implements discretionary access control. Every object has an owner that controls access to that object. Permissions are set in three scopes: user, group, and others. For each scope, permissions to read, write, and execute can be set.

Mandatory Access Control

Mandatory access control (MAC), defined by TCSEC standard, provides more restrictions than DAC. In this type of access control, operating system can prevent subjects from performing operations on objects. This is achieved by attaching subjects and objects set of security attributes. When a subject (usually a process) wants to perform an operation on an object (file, directory, socket, etc.), operating system first examines these attributes. Security policy is then used to determine whether this operation should be allowed or not. When using MAC, users do not have the ability to override the security policy and, for example, propagate sensitive information.

There are several implementations of MAC. Linux kernel currently contains several security modules implemented using *Linux Security Modules* (LSM) framework [6]. Security-Enhanced Linux, developed by National Security Agency and Red Hat [2], is used in Red Hat Enterprise Linux (RHEL), CentOS, Fedora, and Android [10, 9, 7]. AppArmor developed by SUSE, is used in SUSE Linux Enterprise, openSUSE and Ubuntu [5, 1]. There are two other Linux security modules, Smack and TOMOYO Linux.

2.2.3 SELinux and MAC Checks

SELinux security checks are carried out after standard Linux DAC checks. When running an SELinux-enabled system, when a userspace process makes a system call, standard file permissions are checked first. Then, if the access is allowed, the Linux Security Module hooks calls security checks in SELinux. See figure 2.2.

2.2.4 SELinux Users

SELinux uses its own user names that are different from standard Linux user names [14, p. 24]. Every Linux user is associated to an SELinux user. For example, Linux user `root` is mapped to SELinux user `unconfined_u` on Fedora 27. There is a special SELinux user that is not mapped to any user: `system_u`.

Available SELinux users can be listed using the `seinfo --user` command:

```
$ seinfo --user
```

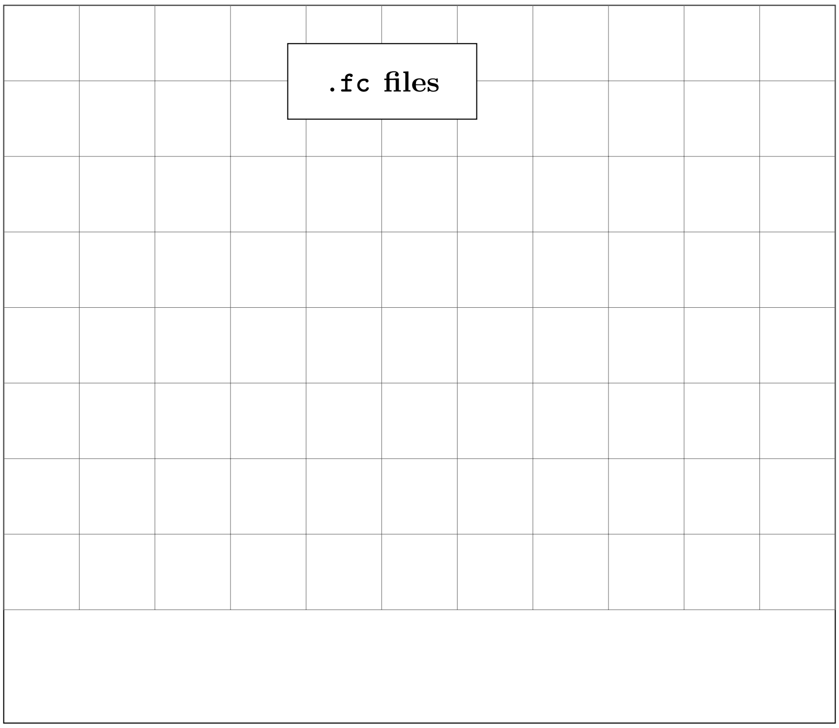

Figure 2.2: DAC and MAC checks.

```
Users: 8
   guest_u
   root
   staff_u
   sysadm_u
   system_u
   unconfined_u
   user_u
   xguest_u
```

2.2.5 Role-Based Access Control

SELinux uses role-based access control (RBAC) as one of its security mechanisms. RBAC as general concept is based on users, roles, and permissions, and relationships between them. RBAC defines the role-permission, user-role, and role-role relationships.

In SELinux, every user is associated to one or more roles [14, p. 24]. Each role can access only types that are associated to that role. For example, user `system_u` is associated to roles `unconfined_r` and `system_r` on Fedora 27.

Available SELinux roles can be listed using the `seinfo --role` command:

```
$ seinfo --role

Roles: 14
   auditadm_r
   dbadm_r
```

```
   guest_r
   logadm_r
   nx_server_r
   object_r
   secadm_r
   staff_r
   sysadm_r
   system_r
   unconfined_r
   user_r
   webadm_r
   xguest_r
```

2.2.6 Type Enforcement

SELinux uses type enforcement for enforcing mandatory access control [14, pp. 25–26]. All subjects and objects have a type associated. Processes running with the same type are called a *domain*. SELinux policy then contains rules that allow domains access types.

Available SELinux types can be listed using the `seinfo --type` command:

```
$ seinfo --type

Types: 4845
   abrt_t
   alsa_t
   antivirus_t
   bin_t
   cluster_t
   crond_t
   ...
```

2.2.7 Multi-Level and Multi-Category Security

In addition to type enforcement and role-based access control, SELinux also supports multi-level security (MLS) and multi-category security (MCS) [14, pp. 48–53]. For the purposes of MLS and MCS, security context is extended by level or range entry.

Security levels conform to the Bell-LaPadula model. For processes, security levels describe subjects clearance, for objects, they describe objects classification. Process running at certain security level can:

- read and write at their current level,
- read only at lower levels,
- write only at higher levels.

This means that processes cannot read data with higher security level and cannot leak sensitive information to the lower levels. Multi-level security is not used by default in most SELinux-enabled Linux distributions such as Fedora or RHEL, but it is supported.

2.2.8 SELinux Security Context

Security decisions are based on a *security context* that must be assigned to every subject and object [14, pp. 27–28]. The security context is sometimes reffered to as *security label* or just *label*. The security context is a string in the following form:

```
user:role:type[:range]
```

Where:

user The SELinux user (see section 2.2.4).

role The SELinux role used by role-based access control (see section 2.2.5).

type The SELinux type used by type enforcement (see section 2.2.6).

range Used by MLS or MCS (see section 2.2.7). It is optional.

Example of subject security contexts:

```
$ ps -eZ
LABEL                              PID TTY          TIME CMD
system_u:system_r:init_t:s0          1 ?        00:00:04 systemd
system_u:system_r:kernel_t:s0        2 ?        00:00:00 kthreadd
system_u:system_r:auditd_t:s0     1139 ?        00:00:00 auditd
system_u:system_r:alsa_t:s0       1164 ?        00:00:00 alsactl
...
```

Example of object security contexts:

```
$ ls -Z /etc
               system_u:object_r:etc_t:s0 alsa
         system_u:object_r:cupsd_etc_t:s0 cups
          system_u:object_r:dhcp_etc_t:s0 dhcp
       system_u:object_r:passwd_file_t:s0 passwd
          system_u:object_r:net_conf_t:s0 resolv.conf
...
```

2.2.9 Object Classes

Each object belongs to an object class. Object classes specify operations that can be performed on the object [14, pp. 29–30]. For example, on Fedora 27, there are the following classes:

```
$ seinfo --class

Classes: 97
   blk_file
   chr_file
   dbus
   dir
   fd
   file
   filesystem
```

```
  ipc
  ...
```

Each class is associated to a set of permissions. For example, on Fedora 27, class `node` provides the following permissions:

```
$ seinfo --class node -x

Classes: 1
   class node
{
        dccp_send
        enforce_dest
        tcp_recv
        rawip_send
        tcp_send
        udp_recv
        dccp_recv
        sendto
        udp_send
        recvfrom
        rawip_recv
}
```

SELinux object classes maps to the kernel object classes (files, sockets, etc.) and userspace objects (for X-Windows or D-Bus).

2.2.10 Labeling Subjects and Objects

Security contexts for subjects and objects are computed by the kernel security server using several policy statements [14, pp. 31–33].

Labeling Processes

The first init process usually transitions to its own unique domain, for example `init_t`. On fork, the child process inherits the security context of its parent. On exec, the child process may transition to different security context. This is achieved by type transition policy statements. SELinux-aware processes may change context by calling `setcon()` or `setexeccon()` functions from the libselinux library.

Labeling Files

Security context for files is computed as follows:

`user` User is inherited from the creating process.

`role` Role defaults to `object_r` unless modified by `role_transition` statement.

`type` Type defaults to the type of the parent directory unless modified by `type_transition` statement.

`range/level` Defaults to the low/current level of the creating process unless modified by `range_transition` statement.

File contexts are covered in depth in section 2.4.

2.2.11 Type Transitions

To run different processes in different domains, we need a way how to *transition* a process from one domain to another. To attach specific label to an object, we need to transition an object from one type to another. Both transitions can be achieved using the `type_transition` statement.

Domain Transition

Starting new process with different security context is called domain transition [14, pp. 43–47]. For example, `systemd` process running as `init_t` needs to start the Apache HTTP Server as `httpd_t`. Apache executables are labeled `httpd_exec_t`. The following policy rule allows the transition:

```
type_transition init_t httpd_exec_t:process httpd_t;
```

The `systemd` process does not need to be aware of SELinux. Because of the `type_transition` rule, the `exec` call will automatically perform the transition. There are conditions that needs to be met before a domain transition can happen:

1. The source domain has permission to transition into the target domain. For example:

   ```
   allow init_t httpd_t:process transition;
   ```

2. The source domain has permission to read and execute the binary. For example:

   ```
   allow init_t httpd_exec_t:file { execute read getattr };
   ```

3. The context of the executable needs to be set as an entry point into the target domain. For example:

   ```
   allow httpd_t httpd_exec_t:file entrypoint;
   ```

Object Transition

When a new object is created it inherits the security context of its parent. When it is required that the object has different context, an object transition must be used [14, pp. 47–48]. For example when an X server creates a file in the `/tmp` directory (which has context `tmp_t`), it gets context `user_tmp_t`. This is achieved by the following `type_transition` rule:

```
type_transition xserver_t tmp_t:file user_tmp_t;
```

The X server does not need to be aware of SELinux, the kernel computes the label automatically.

2.2.12 SELinux Modes of Operation

SELinux has three modes of operation [10]. The default mode is *enforcing*. In this mode, everything which is not allowed by the policy is denied. When a process tries to perform an action which is not allowed by the policy, it is logged. In *permissive* mode, SELinux is not enforcing the policy, it only logs actions. In *disabled* mode, SELinux is turned off.

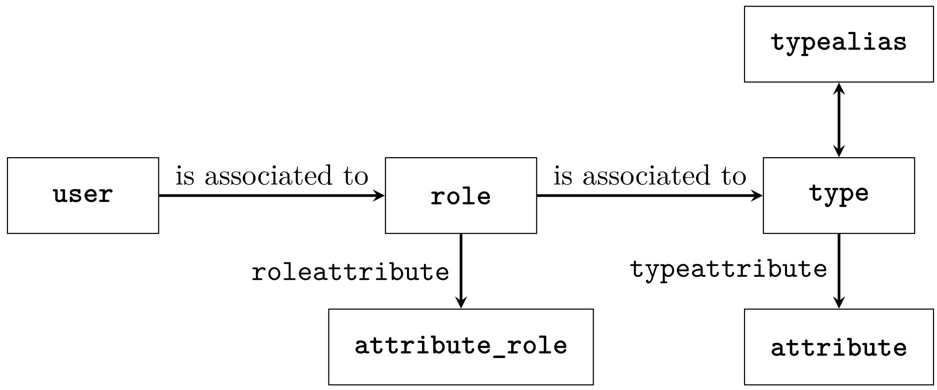

Figure 2.3: Relationship of user, role, and type statements

Running SELinux in permissive is useful for catching AVC denials that can be analyzes using the audit2allow utility. To perform single task (for example to save a file), several SELinux checks are usually needed. When running in enforcing mode, only the first denial would be logged and the troubleshooting would be more difficult.

2.3 SELinux Policy

Security decisions made by the security server in kernel are made using SELinux policy. This section describes the most important SELinux policy statements.

SELinux supports either monolithic (compiled from single source file) or modular policy. Modular policy, which is used in Fedora and RHEL, consists of mandatory base policy source file and loadable modules. In Fedora, almost every module contains policy for one application or service, such as the `apache` or `xserver` module. Some policy statements are valid only in base policy or in policy module.

SELinux policy statements starts with a statement keyword usually followed by several identifiers and semicolon at the end. Comments starts with a "#". Example of an allow rule:

```
# This is an allow rule
allow httpd_t httpd_exec_t: file { ioctl read getattr lock execute open };
```

2.3.1 User, Role and Type Statements

To support mechanisms such as type enforcement, role-based access control, and multi-level and multi-category security, SELinux assigns subjects and objects security contexts. Security context is combination of user, role, type, and optionally range identifiers (see section 2.2.8). This section describes policy statements that declare these identifiers.

SELinux users are declared using the `user` statement. Users are assigned one or more roles. SELinux roles are declared using the `role` statement. Roles are assigned types that they can access. SELinux types are declared using the `type` statement.

Roles can be grouped together using the `attribute_role` and `roleattribute` statements. Types can be grouped together using the `attribute` and `typeattribute` statements. Type aliases can be defined using `typealias` statements. The relationship between various statements is shown in figure 2.3.

User Statements

The `user` statement declares an identifier for an SELinux user. Syntax:

```
user seuser_id roles role_id;
```

Where:

seuser_id SELinux user identifier.

role_id One or more role identifiers.

Example from Fedora 27:

```
user staff_u roles { system_r unconfined_r sysadm_r staff_r };
```

Role Statements

The `role` statement either declares an identifier for an SELinux role and optionally associates a role to one or more types. Syntax:

```
role role_id;
role role_id types type_id;
```

Where:

role_id SELinux role identifier.

type_id One or more type identifiers.

Example from Fedora 27:

```
role auditadm_r types { auditadm_t auditadm_screen_t auditadm_su_t
    auditadm_sudo_t chkpwd_t updpwd_t exim_t auditctl_t auditd_t
    mailman_mail_t user_mail_t postfix_postdrop_t postfix_postqueue_t
    qmail_inject_t qmail_queue_t run_init_t user_tmp_t vlock_t };
```

Type Statements

The `type` statement declares an identifier for an SELinux type. Type identifiers usually ends with '`_t`' to distinguish them from attribute identifiers. Syntax:

```
type type_id;
type type_id, attribute_id;
type type_id alias alias_id;
type type_id alias alias_id, attribute_id;
```

Where:

type_id SELinux type identifier.

alias_id One or more optional aliases declared by the `typealias` statement. Multiple aliases must be enclosed in braces.

attribute_id One or more optional attributes declared by the `attribute` statement. Multiple attributes must be separated by comma.

Example from Fedora 27:

```
type httpd_sys_content_t alias { httpd_fastcgi_content_t
    httpd_httpd_sys_script_ro_t httpd_fastcgi_script_ro_t },
    httpdcontent, httpd_content_type, entry_type, exec_type, file_type,
    non_auth_file_type, non_security_file_type;
```

Other Statements

The `attribute_role` statement declares an identifier for a group of role identifiers. Syntax:

```
attribute_role attribute_id;
```

The `roleattribute` statement associates roles to role attributes. Syntax:

```
roleattribute role_id attribute_id;
```

The `attribute` statement declares an identifier for a group of type identifiers. Syntax:

```
attribute attribute_id;
```

The `typeattribute` statement associates types to attributes. Syntax:

```
typeattribute type_id attribute_id;
```

The `typealias` statement declares type aliases. Syntax:

```
typealias type_id alias alias_id;
```

2.3.2 Access Vector Rules

The access vector rules support type enforcement within SELinux. They control what access do processes get. The `allow` rule grants an access to an object. Syntax:

```
allow source_type target_type:obj_class perm_set;
```

Where:

`source_type` One or more type or attribute identifiers (see section 2.3.1). This field identifies the subject that is performing the operation.

`target_type` One or more type or attribute identifiers. This field identifies the object that is being accessed. When the target type is same as the source type, `self` keyword can be used instead of target type.

`obj_class` One or more object classes (for example `file` or `tcp_socket`).

`perm_set` One or more permissions (for example `read` or `connectto`).

Example:

```
allow httpd_t samba_share_t:file { getattr open read };
```

In this example, processes running as `httpd_t` are allowed to `getattr`, `open`, and `read` files labeled as `samba_share_t`.

There are three other AV rules that follow the syntax pattern of the `allow` rule:

dontaudit Stops auditing (logging) of denials. It is used when the denial is expected to happen and does not cause any issues. The `dontaudit` rules help to keep audit logs clean.

auditallow Audits the event. The `auditallow` rule itself does not allow the operation, so the rule must appear together with standard `allow` rule.

neverallow Compiler statement that stops compilation if this rule is found somewhere in policy. It is used for marking rules that may be unsecure.

Internally, access vectors defined by AV rules are stored in memory as bit arrays that are 32 bits long. Because of this limitation, object classses cannot have more than 32 different permissions. Extended permission AV rules were introduced to overcome this issue.

2.3.3 Extended Permission Access Vector Rules

Since policy version 30, there are extended permission access vector rules that expand the permission sets. Standard access vector rules operates with 32 bit permission sets, extended permission AV rules adds arbitrary number of 256 bit increments. Extended permission AV rules are currently (as of policy version 31) used only for ioctl whitelisting, but they provide generic tool that can be used in future for more granular control over an operation [16].

Syntax of extended permission AV rules [3]:

```
rule_name source_type target_type : obj_class operation xperm_set;
```

Where:

rule_name is one of the following: `allowxperm`, `dontauditxperm`, `auditallowxperm`, or `neverallowxperm`. The meaning is same as with standard AV rules. The `allowxperm` rule allows the access, the `dontauditxperm` rule denies and logs the access, the `auditallowxperm` rule logs the access, and the `neverallowxperm` rules is a compiler statement to prevent unsecure rules from appearing in policy.

source_type, target_type, obj_class are source type, target type, and object class, same as with standard AV rule.

operation is a single keyword defining the operation to be implemented by the rule. As of policy version 31, only the `ioctl` operation is supported. In contrast to permissions in standard access vector rules, each extended permission AV rule has only one operation (standard AV rules can have many permissions).

xperm_set are extended permissions represented by numeric values. The meaning of values depends on the operation. Values can be written in decimal or hexadecimal form, for example `42` or `0x2a`. Multiple values must be separated by space and enclosed in braces, for example `{ 1 2 3 }`. Value ranges are supported, for example `50-60` (both 50 and 60 are included in the range). To allow all values except for those explicitly listed, the complement operator can be used, for example `~{ 1 2 3 }`.

Example of an extended permission AV rule:

```
allowxperm my_app_t my_socket_t : tcp_socket ioctl { 20 30 0x40 50-60 };
```

This rule allows a process running as `my_app_t` to call `ioctl` on a TCP socket labeled `my_socket_t` with parameters 20, 30, 64, or any number from 50 to 60.

Filtering the ioctl System Call

Filtering ioctl calls is as of policy version 31 the only implementation of extended permission AV rules. The ioctl system call accepts three parameters: file descriptor, request number, and a pointer [12]. Extended permission AV rules allows filtering based on the request number. For ioctl calls, the `operation` keyword is `ioctl` and numbers in the `xperm_set` represents request numbers.

When there is only `allow` rule for particular source and target context and object class, all ioctl calls are allowed. With additional `allowxperm` rule, only ioctl calls with parameters allowed by the `allowxperm` rules are allowed. The `allowxperm` rule alone has no effect, for ioctl filtering, both `allow` and `allowxperm` rules must be present.

2.3.4 Policy Modules

The `module` and `require` statements are used to support policy modules. Every policy module must start with the `module` statement. Syntax:

```
module module_name version;
```

Where:

`module_name` Name of the module.

`version` Version number in format `X.Y.Z`.

This name is used to refer to the module when using userspace utilities. For example this command is used to remove module from policy:

```
$ semodule -r module_name
```

The `require` statement indicates what parts of policy are imported from other modules or base policy. Syntax:

```
require { require_list }
```

Where:

`require_list` One or more keywords followed by identifier separated by semicolon. Valid keywords are: `role`, `type`, `attribute`, `user`, `bool`, `sensitivity`, `category`, `class`.

Example of `module` and `require` statements:

```
module my_module 1.2.0;

require {
    type nscd_t, nscd_var_run_t;
    class nscd { getserv getpwd getgrp gethost shmempwd shmemgrp
        shmemhost shmemserv };
}
```

When loading this module, types `nscd_t` and `nscd_var_run_t`, and class `nscd` with specified permissions must be defined somewhere in the policy (either in base policy or in another policy module).

2.3.5 Conditional Policy

SELinux policy allows turning on and off set of policy statements without the need for reloading policy. Conditional policy is defined using the `bool` statement that defines a condition. Then a `if/else` construct is used to mark statements that depends on the condition. Example:

```
bool allow_execmem false;

if (allow_execmem) {
    allow sysadm_t self:process execmem;
}
```

Booleans can be turned on and off using the `semanage boolean` command.

2.3.6 Labeling Network Objects

SELinux policy supports labeling of the following network objects:

Network ports TCP or UDP port numbers.

Network nodes Nodes represented by IP addresses and subnet masks.

Network interfaces Interfaces managed by `ifconfig` (e.g. `eth0`).

Network Interfaces

The `netifcon` statement labels network interface statements. Syntax:

```
netifcon netif_id netif_context packet_context
```

Where:

`netif_id` Name of the network interface (e.g. `eth0`).

`netif_context` Security context of the interface.

`packet_context` Security context of the packets. This is context is not currently used (kernel does not support labeling of packets).

Example:

```
netifcon eth0 system_u:object_r:netif_t:s0 system_u:object_r:netif_t:s0
```

Network Nodes

The `nodecon` statement labels network addresses. Syntax:

```
nodecon subnet netmask node_context
```

Where:

`subnet` The IP address.

`netmask` The subnet mask.

`node_context` Security context of the node.

Example:

```
nodecon ff00:: ff00:: system_u:object_r:multicast_node_t:s0
```

Network Ports

The portcon statement labels TCP and UDP ports. Syntax:

```
portcon protocol port_number port_context
```

Where:

protocol Either udp or tcp.

port_number Port number or a range.

port_context Security context of the port.

Example:

```
portcon tcp 22 system_u:object_r:ssh_port_t:s0
```

2.4 File Contexts

When accessing files, SELinux relies on labels stored with those files to make a security decision. SELinux labels can be viewed using the ls -Z command:

```
$ ls -Z
unconfined_u:object_r:user_home_t:s0    testdir
unconfined_u:object_r:user_home_t:s0    testfile
```

Labels are stored in *extended attributes* in the security namespace [13]. Extended attributes associated with a file can be viewed using the getfattr command:

```
$ getfattr -n security.selinux testfile
# file: testfile
security.selinux="unconfined_u:object_r:user_home_t:s0"
```

2.4.1 Temporary Changes

The chcon command changes the SELinux context of files [10]. User must have the permission to relabel files. The changes made by chcon are overwritten by a file system relabel or running of restorecon.

2.4.2 Type Transition

There are rules in policy that specifies the context of files created by processes. For example, when process running with the httpd_t context creates a file in directory with the var_run_t context, the file will get context httpd_var_run_t:

```
type_transition httpd_t var_run_t:file httpd_var_run_t;
```

Type transitions are explained in section 2.2.11.

2.4.3 File Context Configuration Files

There are situations when files get label that is different than the default one:

1. When moving files, label is preserved. This does not happen when copying files because new file is always created.

2. When SELinux is disabled, labels are not assigned to files.

3. When policy is changed (for example when a module is unloaded), there may be some files left with type that is no longer defined in policy.

For these situations, there is a `file_contexts` file which specifies default contexts for every file based on its path. For example:

```
/run/.*          --  system_u:object_r:var_run_t:s0
/var/.*          --  system_u:object_r:var_t:s0
/etc/.*          --  system_u:object_r:etc_t:s0
/lib/.*          --  system_u:object_r:lib_t:s0
/usr/.*\.cgi     --  system_u:object_r:httpd_sys_script_exec_t:s0
/root(/.*)?      --  system_u:object_r:admin_home_t:s0
/dev/[0-9].*     -c  system_u:object_r:usb_device_t:s0
/dev/.*tty[^/]* -c  system_u:object_r:tty_device_t:s0
```

The `--` means that the context should be applied to all file types (e.g., files, directories, sockets). The `-c` means that the context should be applied only when the file is a character device. Utilities such as restorecon and setfiles uses the `file_contexts` configuration file to relabel files on the filesystem.

2.4.4 Building File Context Configuration Files

Utilities such as restorecon and setfiles uses several files to restore default contexts of files [14, pp. 165–167]:

`file_contexts` Contains default contexts for files.

`file_contexts.homedirs` Contains default contexts for files inside user home directories.

`file_contexts.local` Contains local modifications of default file contexts.

`file_contexts.subs` and `file_contexts.subs_dist` Contains file name substitutions. For example, these files can specify that `/usr/lib64` should be treated the same way as `/usr/lib`.

These files are created when building policy, see figure 2.4. All `.fc` files from base policy and from policy modules are used to build the `file_contexts.template` file. This file may contain rules that has special keywords inside their path, such as `HOME_ROOT`, `HOME_DIR`, or `USER`. All rules without special keywords are used to build the `file_contexts` file used directly by utilities such as restorecon or setfiles.

Rules with special keywords are used to build the `homedir_template` file. These rules are asssociated with user home directories and need to be expanded for individual users using the `genhomedircon` utility. For example the following `homedir_template` entry:

```
HOME_DIR/\.ssh(/.*)?        system_u:object_r:ssh_home_t:s0
```

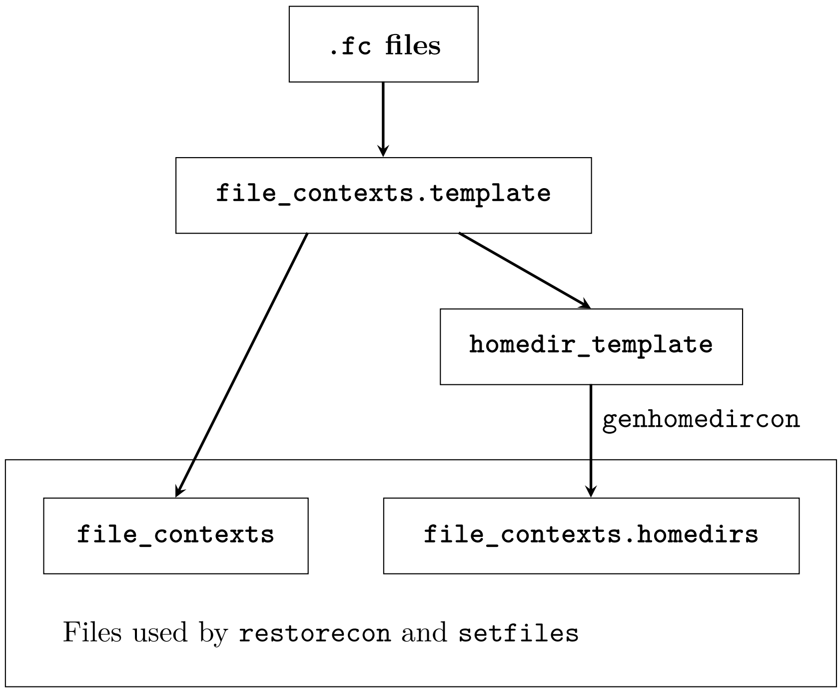

Figure 2.4: File Context Files

would be expanded to the following rules:

```
/home/[^/]*/\.ssh(/.*)?     system_u:object_r:ssh_home_t:s0
/root/\.ssh(/.*)?           system_u:object_r:ssh_home_t:s0
```

Expanded rules are then stored in the `file_contexts.homedirs` file and used by restorecon and setfiles utilities [14, pp. 134–140].

2.4.5 Changing File Context Configuration Files

The `file_contexts.local` file can be changed using the `semanage fcontext` command [10]. For example:

```
# semanage fcontext -a -t samba_share_t /etc/myfile
# semanage fcontext -l -C
SELinux fcontext    type        Context
/etc/myfile         all files   system_u:object_r:samba_share_t:s0
```

In this example, new file contexts entry was added. The rule states that file `/etc/myfile` should obtain context `system_u:object_r:samba_share_t:s0`.

2.5 Auditing Security Events

The *Linux Audit system* provides an auditing system for tracking security-relevant system events. It is used to track file access, monitor system calls, record commands run by user, record failed login attempts and others [8]. The Linux Audit system does not provide additional security by itself, it can be only used to discover security violations.

The Linux Audit system consists of kernel and userspace part. Kernel filters events and sends them to the *audit daemon*. Audit daemon then writes the received events to log file.

There are several userspace tools used for interacting with the audit system and for working with the log file.

2.5.1 Audit and SELinux

In Fedora and RHEL, SELinux uses the Linux Audit system to log security events. When a process tries to perform operation without the permissions, an *Access Vector Cache* (AVC) denial message is logged using the audit daemon [10]. This message can be then processed by tools such as setroubleshoot or audit2allow.

Every AVC message contains information about source context (the context of the process), object class (for example file), and target context (the context of the object). For example, when a process `httpd` running in context `unconfined_u:system_r:httpd_t:s0` is trying to perform the `getattr` operation on file `/var/www/html/file1` with context `system_u:object_r:samba_share_t:s0` and fails, the following AVC message is generated:

```
type=AVC msg=audit(1223024155.684:49): avc:  denied  { getattr }
for pid=2000 comm="httpd" path="/var/www/html/file1" dev=dm-0
ino=399185 scontext=unconfined_u:system_r:httpd_t:s0
tcontext=system_u:object_r:samba_share_t:s0 tclass=file
```

2.6 Troubleshooting SELinux

When SELinux denies access that is requested by a process, the process may fail to function normally and reports error or crashes. Determining if the failure is related to SELinux is done by switching whole SELinux or just one domain into permissive mode. For example, for debugging `httpd` it is advised to set the `httpd_t` domain into permissive mode:

```
# semanage permissive -a httpd_t
```

SELinux denials caused by the `httpd_t` domain would still be logged but not enforced.

On Fedora and RHEL, SELinux denials are analyzed by the `setroubleshootd` daemon that provides suggestions for resolving the problem using various plugins. Majority of problems are caused by missing `allow` rules in policy. To add missing rules, the audit2allow utility is used. However, not every problem requires new policy rules, some issues may require relabeling of objects using restorecon or semanage.

2.7 The audit2allow Utility

The audit2allow is a userspace tool that scans the AVC messages and generates SELinux policy snippets based on them.

2.7.1 Purpose of audit2allow

The audit2allow utility is designed both for system administrators and SELinux policy developers. System administrators use audit2allow to analyze SELinux denials and to create new policy modules. When suitable, the audit2allow utility suggests other options to resolve denials, such as turning on a boolean (see section 2.3.5).

Policy developers use audit2allow to create basis for new policy modules for their products. When writing policy for their program, they can run the programs test suite in permissive mode, collect AVC denials, create policy module, and then manually finish the policy module. Policy developers can use the `--reference` option to generate policy using macros.

2.7.2 Basic Mode of Operation

In default mode, audit2allow scans AVC denial messages and generates policy rules which allow operations that were denied. For example, when the `httpd` process tries to perform the `getattr` operation on the `/var/www/html/file1` file, the following AVC message is generated:

```
type=AVC msg=audit(1223024155.684:49): avc:  denied  { getattr }
for pid=2000 comm="httpd" path="/var/www/html/file1" dev=dm-0
ino=399185 scontext=unconfined_u:system_r:httpd_t:s0
tcontext=system_u:object_r:samba_share_t:s0 tclass=file
```

The audit2allow utility would generate the following policy rule:

```
allow httpd_t samba_share_t:file getattr;
```

The audit2allow utility is able process multiple AVC denial messages, deal with duplicates, and output all rules based on fields in AVC denial messages.

2.7.3 Command-Line Options

The audit2allow utility is able to read AVC messages from stdin, dmesg, audit log, or arbitrary file (see `--dmesg`, `--all`, and `--input` options). There is `--boot` option which loads only messages generated since last boot and `--lastreload` option which loads only messages since last SELinux policy reload.

The audit2allow utility can output the policy rules directly to stdout or file, or create a policy module which can be loaded directly into the policy (see `--module`, `-M`, and `--output` options).

The audit2allow utility is using currently loaded policy (or any other policy specified with the `--policy` option) to get more information about the denials. For example, audit2allow suggests turning on a boolean that would allow the denied operations.

When run with the `--reference` option, audit2allow tries to match the denials against defined interfaces. Example of audit2allow output without the `--reference` option:

```
#============= httpd_t ==============
allow httpd_t samba_share_t:file getattr;
```

Example of audit2allow output with the `--reference` option:

```
require {
        type httpd_t;
}

#============= httpd_t ==============
samba_read_share_files(httpd_t)
```

The audit2allow found an interface which contained the same allow rule. Interfaces creates more readable code but can contain more rules that are necessary.

The `--why` option does not output any policy rules but provides a text description of why the access was denied. Example of `audit2allow --why` output:

```
type=AVC msg=audit(1223024155.684:49): avc: denied { getattr }
for pid=2000 comm="httpd" path="/var/www/html/file1" dev=dm-0
ino=399185 scontext=unconfined_u:system_r:httpd_t:s0
tcontext=system_u:object_r:samba_share_t:s0 tclass=file

    Was caused by:
        Missing type enforcement (TE) allow rule.

        You can use audit2allow to generate a loadable module
        to allow this access.
```

The `--dontaudit` option generates `dontaudit` rules instead of `allow` rules (see section 2.3.2).

2.7.4 How Does audit2allow Work

The audit2allow first collects audit messages from various sources. Messages are stored based on their type and then parsed. Every AVC denial message is analyzed together with binary policy file to find out the reason of denial.

From AVC denial messages, source contexts, target contexts, object classes, and permissions are extracted and converted into *access vector sets*. Each access vector in the set contains unique combination of source context, target context, and object class. Permissions from multiple AVC messages are merged into one access vector set. Example of an access vector set:

```
{
  {
    src: 'unconfined_u:system_r:httpd_t:s0',
    tgt: 'system_u:object_r:samba_share_t:s0',
    cls: 'file', perms: [ 'getattr', 'open' ]
  },
  {
    src: 'unconfined_u:system_r:httpd_t:s0',
    tgt: 'system_u:object_r:sssd_conf_t:s0',
    cls: 'file', perms: [ 'getattr' ]
  }
}
```

Each access vector is then converted into an allow rule. All rules are printed to the output. Example:

```
allow httpd_t samba_share_t:file { getattr open };
allow httpd_t sssd_conf_t:file getattr;
```

The `module` and `require` statements (see section 2.3.4) may be optionally written to the output. Various other information is stored during processing. The audit2allow prints comments with helpful messages.

2.7.5 Implementation of audit2allow

The audit2allow utility is part of SELinux userspace. It is written mostly in Python with several parts written in C. It uses sepolgen and sepolicy Python packages and libselinux and libsepol libraries.

Main script, `audit2allow`, parses command-line options, retrieves audit messages, and prints the output. Main logic of converting AVC denial messages to access vector rules is implemented in package sepolgen.

The sepolgen package contains the following modules:

`audit.py` Defines classes for various audit messages, contains audit message parser.

`access.py` Defines access vectors and access vector sets.

`policygen.py` Creates policy rules based on access vectors.

`refpolicy.py` Contains classes that represent the policy statements.

`output.py` Outputs the generated rules.

Other modules There are several other modules which are either not significant (e.g. the `utils.py` module) or used only for generating policy using interfaces (e.g. the `interfaces.py` package).

The audit2allow Script

The main script does the following steps:

1. Parse command-line arguments and check potential conflicts.
2. Read audit messages. Create `AuditParser` instance and feed it the messages.
3. Filter the messages (if specified by the `--type` option) and convert them to access vectors.
4. Create and setup a `PolicyGenerator` instance, feed it the access vectors, and convert them to policy rules.
5. Write the output.

The audit.py Module

The `audit.py` module is used for parsing audit messages. It is not a general purpose audit parsing library, it is meant to parse mainly AVC messages and policy load messages.

The `AuditParser` class reads strings and creates objects of appropriate type for each message. The `AuditMessage` class is the base class for all message types. The `AVCMessage` class represents AVC denials and is used for generating access vectors.

After parsing of AVC message, the denial is analyzed in `audit2why.c` module (from libselinux library). The `audit2why.c` module tries to find out the reason of the denial by analyzing the policy. The module is written in C and uses the libsepol library.

Each message is then converted to an access vector from the `access` module. AVC denial messages can be filtered using regular expressions via the `AVCTypeFilter` class. Only messages that match the regular expression are processed.

Policy load messages are important for the `--lastreload` command-line option. The `AuditParser` then processes only messages after last policy load message.

The access.py Module

The `access.py` module defines the `AccessVector` and `AccessVectorSet` classes. Access vector is a basic representation of an access in SELinux. It contains single source and target type, single object class, and a set of permissions. Every AVC denial message can be converted into an access vector. For example this AVC denial message:

```
type=AVC msg=audit(1223024155.684:49): avc:  denied  { getattr }
for pid=2000 comm="httpd" path="/var/www/html/file1" dev=dm-0
ino=399185 scontext=unconfined_u:system_r:httpd_t:s0
tcontext=system_u:object_r:samba_share_t:s0 tclass=file
```

would be converted into the following access vector:

```
{
    source_context: 'unconfined_u:system_r:httpd_t:s0',
    target_context: 'system_u:object_r:samba_share_t:s0',
    object_class: 'file',
    permissions: [ 'getattr' ]
}
```

Multiple access vectors are aggregated in access vector sets. Access vectors that share the same source and target type and object class are merged together so that there are no duplicates. For example, if we add the following access vector:

```
{
    source_context: 'unconfined_u:system_r:httpd_t:s0',
    target_context: 'system_u:object_r:samba_share_t:s0',
    object_class: 'file',
    permissions: [ 'open', 'read' ]
}
```

to the access vector above, they would be merged into the following access vector (they share source and target context and object class):

```
{
    source_context: 'unconfined_u:system_r:httpd_t:s0',
    target_context: 'system_u:object_r:samba_share_t:s0',
    object_class: 'file',
    permissions: [ 'getattr', 'open', 'read' ]
}
```

Access vector sets serve as a basis for generating policy access vector rules in the `policygen.py` module.

The policygen.py Module

The `policygen.py` module defines the `PolicyGenerator` class that generates policy module from access vectors. `PolicyGenerator` converts access vector set into SELinux policy statements. For example, this access vector:

```
{
    source_context: 'unconfined_u:system_r:httpd_t:s0',
    target_context: 'system_u:object_r:samba_share_t:s0',
```

```
    object_class: 'file',
    permissions: [ 'getattr', 'open', 'read' ]
}
```

would be converted into the following policy statement:

```
allow httpd_t samba_share_t:file { getattr open read };
```

PolicyGenerator uses objects from the refpolicy.py module to represent policy statements. PolicyGenerator provides several configuration methods:

set_gen_refpol() Turn on interface generation.

set_gen_requires() Generate require statements that are neccessary for creating a standalone policy module (see section 2.3.4).

set_gen_explain() Add comments explaining why were the policy statements generated.

set_gen_dontaudit() Generate dontaudit rules instead of allow rules (see section 2.3.2).

The output of PolicyGenerator is a tree-like structure containing generated policy statements. The output.py module then just prints out every statement.

The refpolicy.py Module

This module contains classes that represent SELinux policy statements. The Node and Leaf classes are base classes for all policy statements. Every statement is either a node that is a parent of other statements (for example the Module class), or a leaf (for example the AVRule class). The refpolicy.py module contains functions for traversing trees made out of nodes and leaves. These functions are used when printing statements in the output.py module.

The IdSet class represents set of arbitrary identifiers and is used by many statements for storing permissions and other sets. The SecurityContext class represents an SELinux security context. Classes such as TypeAttribute, RoleAttribute, Role, Type, and others represent policy statements as described in section 2.3 and are used mainly for interface generation.

For basic operation mode, the following classes are used: AVRule, ModuleDeclaration, Module, and Require. The AVRule class contains the following attributes:

src_types IdSet() of source types.

tgt_types IdSet() of target types.

obj_classes IdSet() of object classes.

perms IdSet() of permissions.

rule_type One of the following: ALLOW, DONTAUDIT, AUDITALLOW, or NEVERALLOW.

Class Module serves only as a node that is parent to all statements inside a module. Class ModuleDeclaration represents the module statement and is generated with the --module option. Class Require represent the require statement inside policy modules and is generated with either --module or --require options.

Chapter 3

Analysis

Several improvements to audit2allow were proposed:

1. Changing label of an object instead of creating new policy rules. This includes checking of mislabeled files, labeling network ports, nodes, and interfaces.
2. Support for new SELinux policy statements.

3.1 Extended Permission Access Vector Rules

Since policy version 30, SELinux policy supports extended permission access vector rules (see section 2.3.3). Usage of extended permission AV rules introduces situations when audit2allow is not able to detect the true cause of denial. As a result, when using extended permission AV rules, audit2allow may suggest rules that do not solve the denial.

3.1.1 AVC Denials Caused by Extended Permission AV Rules

Suppose there are following rules present in the policy:

```
allow src_t tgt_t : tcp_socket ioctl;
allowxperm src_t tgt_t : tcp_socket ioctl 0x42;
```

When the process tries to call `ioctl(0x1234, ...)`, the operation would be denied, because only syscall `ioctl(0x42, ...)` is allowed. The following AVC denial message would be generated:

```
type=AVC msg=audit(1515017775.689:1722): avc:  denied  { ioctl } for
pid=14587 comm="test" dev="dm-0" ino=8390105 ioctlcmd=0x1234
scontext=unconfined_u:unconfined_r:src_t:s0-s0:c0.c1023
tcontext=unconfined_u:object_r:tgt_t:s0 tclass=tcp_socket permissive=0
```

The `ioctlcmd` field contains first parameter of ioctl syscall that was denied. This value can be used to construct an allowxperm rule to allow this operation.

When used for troubleshooting this AVC denial, audit2allow produces the following output:

```
#============= src_t ==============

#!!!! This avc is allowed in the current policy
allow src_t tgt_t:tcp_socket ioctl;
```

which is not helpful. User must know about extended permissions and assume that the allow rule was overridden.

3.1.2 Generating Extended Permission AV Rules in audit2allow

The audit2allow does have all the information to generate extended permission AV rules. There are two situations that may arise when using extended permission AV rules:

- There is neither `allow` nor `allowxperm` rule in the policy. The audit2allow utility has two options: either generate only allow rule (current behaviour) or generate `allow` and `allowxperm` rules. Generating `allowxperm` rules may be inefficient for many processes, because they use lot of different ioctl calls.
- There is both `allow` and `allowxperm` rule in the policy. This means that the specific ioctl parameter is not allowed. In this case `allowxperm` rule should be generated.

It is not possible to distinguish these two sitautions by analyzing the AVC denial itself, because both denials contain the `ioctlcmd` field. The audit2allow utility would need to analyze the binary policy. The audit2allow utility may rather generate extended permission AV rules in all cases (stricter, more secure solution) or only when requested by user (for example using command-line option, less secure solution, does not break backward compatibility).

In case of using command-line option, there is still a risk, that the user does not know that he or she should be using that option. Consider the following example:

```
#============= src_t ==============

#!!!! This avc is allowed in the current policy
allow src_t tgt_t:tcp_socket ioctl;
```

In this example, it is not clear, that the denial is caused by extended permission AV rule. The audit2allow utility should generate an explanation in situations, when the access would be allowed and the AVC denial messages contains the `ioctlcmd` field. For example:

```
#============= src_t ==============

#!!!! This avc is allowed in the current policy
#!!!! This av rule may have been overriden by extended permission av rule
allow src_t tgt_t:tcp_socket ioctl;
```

3.2 Mislabeled Files

SELinux relies on files that are correctly labeled. Sometimes, files get mislabeled, processes cannot access these files and causes AVC denials. When used for troubleshooting, audit2allow suggests adding new rules to the policy instead of changing label of the file.

3.2.1 AVC Denial Messages Caused by Mislabeled Files

When a process is trying to access file that is mislabeled, the operation is usually denied (unless the process has access also to the incorrect label). For example, when user moves content from `/root` directory to the `/var/www/html/` directory, files retain their original label:

```
$ ls /var/www/html
unconfined_u:object_r:httpd_sys_content_t:s0 index.html
       unconfined_u:object_r:admin_home_t:s0 my_file.html
```

File `index.html` has correct label, but file `my_file.html` has incorrect label. The `httpd` process cannot access files labeled `admin_home_t`, because there are no allow rules in the policy for this operation:

```
$ sesearch -A -s httpd_t -t admin_home_t -c file -p read
(nothing)
```

As a result, when trying to view `my_file.html`, similar AVC denial message is generated:

```
type=AVC msg=audit(1226270358.848:238): avc:  denied  { read }
for  pid=13349 comm="httpd" ino=8390105 name="my_file.html"
dev=dm-0 ino=218171 scontext=system_u:system_r:httpd_t:s0
tcontext=system_u:object_r:admin_home_t:s0 tclass=file
```

In this case, there is the `ino` field, which contains inode number associated with the denial, and the `name` field, which contains name of the file (but not the full path). In cases of `getattr` denial, the `path` field is present. AVC denial may also happen because a process cannot access list of files in directory.

3.2.2 Solving Problems With Mislabeled Files

The restorecon utility uses file contexts files to get default security contexts of files (see section 2.4). For example, when file `/var/www/html/my_file.html` is mislabeled, the denial should be fixed by running restorecon on the file:

```
# restorecon -v /var/www/html/my_file.html
Relabeled /var/www/html/my_file.html from unconfined_u:object_r:admin_home_t:s0
to unconfined_u:object_r:httpd_sys_content_t:s0
```

When using audit2allow, the following rules are generated:

```
allow httpd_t admin_home_t:file getattr;
```

This means that the `httpd` process would gain access to all root's files. This solution is not secure because it is adding unnecessary rules to the policy and it does not solve the real problem.

3.2.3 Improving audit2allow

The audit2allow utility should detect when the AVC message is caused by mislabeled file and suggest solution using the restorecon utility.

There are three fields in the AVC message that can be used to detect if the file was mislabeled: `path`, `name` and `inode`. When the `path` field is present, audit2allow can run `matchpathcon` to get the default context of the file and compare it with actual file context.

In many cases, the `path` field is not present, only inode number and name of the file (without full path). In this case it is difficult to find the full path of the file. Ryan Hallisey created solution [15] that is using `locate` utility to get all files matching the name and then stat these files to get the inode number. This solution is only partial, it does not work on files that are not indexed in the database created by `updatedb`.

3.3 Labeling Network Ports, Nodes, and Interfaces

SELinux policy supports labeling of TCP and UDP ports, network nodes (represented by IP addresses and subnet masks), and network interfaces (e.g. `eth0`).

3.3.1 Network Ports

SELinux can enforce binding to system ports. For example, in Fedora 27, there are several hundred `portcon` rules that label TCP and UDP ports. Example:

```
$ seinfo --portcon

Portcon: 615
   portcon tcp 1-511 system_u:object_r:reserved_port_t:s0
   portcon tcp 7 system_u:object_r:echo_port_t:s0
   portcon tcp 21 system_u:object_r:ftp_port_t:s0
   portcon tcp 22 system_u:object_r:ssh_port_t:s0
   portcon tcp 53 system_u:object_r:dns_port_t:s0
   portcon tcp 80 system_u:object_r:http_port_t:s0
   portcon udp 1-511 system_u:object_r:reserved_port_t:s0
   portcon udp 1 system_u:object_r:inetd_child_port_t:s0
   portcon udp 7 system_u:object_r:echo_port_t:s0
   portcon udp 53 system_u:object_r:dns_port_t:s0
   portcon udp 67 system_u:object_r:dhcpd_port_t:s0
   ...
```

Portcon rules can overlap, for example TCP port number 80 is labeled `http_port_t` but also `reserved_port_t` because it is in range 1–511. Every port has either domain-specific label or one of the following labels (based on range):

1–511	`reserved_port_t`
512–1023	`hi_reserved_port_t`
1024–32767	`unreserved_port_t`
32768–61000	`ephemeral_port_t`
61001–65535	`unreserved_port_t`

When a process tries to bind port and it is denied by policy, AVC message is generated. For example:

```
type=AVC msg=audit(1516026512.648:4191): avc:  denied  { name_bind } for
pid=6116 comm="test" src=43 scontext=unconfined_u:unconfined_r:my_app_t:s0
tcontext=system_u:object_r:reserved_port_t:s0
tclass=tcp_socket permissive=0
```

Proper way how to allow the process to bind on port number 43 would be to label this port with a application-specific context. The audit2allow suggest adding the following rule to the policy:

```
allow my_app_t reserved_port_t:tcp_socket name_bind;
```

This rule would allow `my_app_t` access to all reserved ports which is unneccessary and potentially unsecure.

Ports can be labeled using the `portcon` rules, but as of policy version 31, these rules are not valid in policy module, only in base policy. So audit2allow would not be able to generate `portcon` rules directly. Another way of labeling ports is via the `semanage port` command. The audit2allow should suggest using the `semanage port` command when appropriate.

3.3.2 Network Nodes

SELinux is capable of labeling network nodes. For example, there can be rules that allow process to communicate only on private LAN or even only on local host. Attempts to violate these rules would then produce AVC denial messages that contain IP address of the node.

Proper solution would be to modify label of certain subnet of the network, for example using the `semanage node` command. AVC denial messages provides only IP addresses. As IP addresses can change often, labeling single network node would not be useful. Labeling of network nodes is not used on Fedora and RHEL. Improving the audit2allow utility to suggest `semanage node` commands would not make a big impact on security.

3.3.3 Network Interfaces

SELinux is capable of labeling network interfaces (such as `eth0`). In Fedora and RHEL, labeling of network interfaces is not used. Improving audit2allow to suggest `semanage interface` commands would not make a big impact on security.

Chapter 4

Implementation

From the list of possible improvements to audit2allow, the following improvements were implemented:

- Support for extended permissions. The audit2allow utility can now detect denials that may be caused by extended permission AV rules. With the `--xperms` option, audit2allow generates extended AV rules.
- Checking mislabeled files. The audit2allow utility now parses the `path` field in AVC denial messages and checks if files have default context. When the context in AVC denial message is different than the default one, audit2allow produces warning.

4.1 Extended Permissions

Main script `audit2allow` and modules `audit.py`, `access.py`, `policygen.py`, `refpolicy.py` were modified to support extended permissions. Support for extended permission AV rules with the `--reference` option was not implemented.

In main script, new command-line option `--xperms` was added to turn on generating of the extended permission AV rules. This option turns on extended permission AV rules generating in `PoligyGenerator`.

4.1.1 Parsing AVC Denial Messages

Despite extended permissions being a general concept, AVC denial messages generated by the Linux Audit system contain operation-specific field `ioctlcmd`. In the future, with introduction of new operation to the extended permissions, the `audit.py` module will need to be updated.

The `audit.py` module was extended to parse the `ioctlcmd` field in AVC denial messages. The `ioctlcmd` field is then converted to fit the general concept of extended permissions and passed to the access vector set. The following code in the `AuditParser.to_access()` method converts operation-specific fields to generic extended permission dictionary:

```
if avc.ioctlcmd:
    xperm_set = refpolicy.XpermSet()
    xperm_set.add(avc.ioctlcmd)
    xperms = { "ioctl": xperm_set }
```

4.1.2 Storing Extended Permissions

AVC denial messages are converted to access vectors (see section 2.7.5). Two messages that contains same source and target context and object class are merged together. Denials caused by the ioctl system call produce AVC denial messages containing the `ioctlcmd` field. These messages can be treated the same way as any other AVC denial message, but for generating extended permission AV rules, the `ioctlcmd` field must be stored. The `AccessVector` class was extended to store the `ioctlcmd` field and potentionally any other extended permission fields introduced in the future.

Single access vector may be product of several AVC denial messages that contain different values in the `ioctlcmd` field. For the purpose of representing extended permission values, new class `XpermSet` was implemented in the `refpolicy.py` module. The `XpermSet` class supports merging of multiple values. It is likely that in the future new extended permission operation will be introduced. Access vector must store values for different operations. For this purpose, `XpermSet` objects are stored in a dictionary where the operation is the key, for example:

```
{
    'ioctl': <refpolicy.XpermSet() object>,
    'other_command': <refpolicy.XpermSet() object>,
    'another_command': <refpolicy.XpermSet() object>,
}
```

When merging two access vectors, the permissions set are merged using the `union()` method. Extended permission dictionary is merged using the following code from the `AccessVector.merge()` method:

```
for op in av.xperms:
    if op not in self.xperms:
        self.xperms[op] = refpolicy.XpermSet()
    self.xperms[op].extend(av.xperms[op])
```

The `av` variable contains access vector that is to be merged with `self` access vector.

4.1.3 Representation of Extended Permission AV Rules

Extended permission access vector rules are represented by new `AVExtRule` class from the `refpolicy.py` module. This class is very similar to the `AVRule` class, but contains attributes specific to extended permission AV rules. The `operation` attribute is a string identifying the operation, e.g. 'ioctl'. The `xperms` attribute is an `XpermSet` instance. Method `to_string()` prints out the rule. Example of an extended permission AV rule:

```
allowxperm my_app_t my_socket_t : tcp_socket ioctl { 20 30 0x40 50-60 };
```

The `AVExtRule` class is initialized from an access vector using the `from_av()` method. This method contains the `op` parameter to select which extended permission operation from the access vector should appear in the rule.

Without extended permissions, every access vector can be converted into single AV rule. With extended permissions attached to the access vector, to fully convert access vector to policy rules, there needs to be one AV rule and possibly several extended permission AV rules. For example, this access vector:

```
{
```

```
    source_context: 'unconfined_u:system_r:httpd_t:s0',
    target_context: 'system_u:object_r:samba_share_t:s0',
    object_class: 'file',
    permissions: { 'getattr', 'ioctl', 'open' }
    extended_permissions: {
        'ioctl': { 1, 2, 3 },
        'other_command': { 40, 50, 60 },
        'another_command': { 700, 800, 900 },
    }
}
```

would be converted into these policy rules[1]:

```
allow httpd_t samba_share_t:file { getattr ioctl open };
allowxperm httpd_t samba_share_t:file ioctl { 1 2 3 };
allowxperm httpd_t samba_share_t:file other_command { 40 50 60 };
allowxperm httpd_t samba_share_t:file another_command { 700 800 900 };
```

4.1.4 Generating Extended Permission AV Rules

New configuration method `set_gen_xperms()` was added to the `PolicyGenerator` from the `policygen.py` module, to specify whether the extended permission AV rules should be generated.

In the old implementation, method `add_access()` called the `__add_allow_rules()` method which generated AV rules for every access vector set. In new implementation, there are two methods, `__add_av_rule()` and `__add_ext_av_rules()`. Both accept access vector as a parameter, the `__add_av_rule()` method generates single standard access vector rule, `__add_ext_av_rules()` method generates (possibly) several extended permission access vector rules. Both methods are called from the `add_access()` methods:

```
for av in raw_allow:
    self.__add_av_rule(av)
    if self.xperms and av.xperms:
        self.__add_ext_av_rules(av)
```

4.2 Mislabeled Files

The audit2allow utility was extended to check the default context of file if the `path` field is present in the AVC denial message. The `audit.py` and `policygen.py` modules were modified.

4.2.1 Parsing Path

The `audit.py` module was modified to parse the `path` field in AVC denial messages. Only paths found directly in AVC denial messages will be analyzed later by matchpathcon. As a result, the context will not be checked in many cases, because the AVC denials often does not contain full path, only inode number and file name.

[1] Note that as of policy version 31, only the `ioctl` operation is supported, operations `other_command` and `another_command` were added only as an example.

4.2.2 Checking Default Context

In `policygen.py` module, new option was added to the `PolicyGenerator` to turn on or off checking of mislabeled files. Checking is turned on by default. Every AVC message from every access vector is checked whether it contains the path. Default context of the path is then obtained via `selinux.matchpathcon()` function. Target context of the access vector is then compared with default context. In case of difference, comment is added to warn user about mislabeled file. For example:

```
#============= src_t ==============

#!!!! The '/etc/myfile' file has other than default context
allow src_t tgt_t:file getattr;
```

Chapter 5

Functional Testing

The functionality of implemented features to audit2allow was tested by extending existing unit tests and writing integration tests that are focused on interoperation between audit2allow, SELinux, and Linux Audit system.

5.1 Testing Extended Permissions Implementation

Unit tests were extended to ensure that the new functionality does not break existing code. New test cases were added to test the new features.

5.1.1 Testing audit2allow Script

In main script, new `--xperms` and `-x` option was added to turn on extended permission AV rules generation.

`Audit2allowTests` class from the `test_audit2allow.py` module:	
`test_xperms()`	Test that `audit2allow -x` produces at least one `allowxperm` rule.

5.1.2 Testing audit.py Module

In this module, audit message parser was modified to recognize new field in AVC denial messages and to convert the field to extended permissions.

Testing `AVCMessage.__init__()`:

`TestAVCMessage` class from the `test_audit.py` module:	
`test_defs()`	Test that `AVCMessage.ioctlcmd` is `None`.

Testing `AVCMessage.from_split_string()`:

`TestAVCMessage` class from the `test_audit.py` module:	
`test_xperms()`	Test that the `ioctlcmd` field is parsed.
`test_xperms_invalid()`	Test message with invalid value in the `ioctlcmd` field
`test_xperms_without()`	Test message without the `ioctlcmd` field.

Testing `AVCMessage.to_access()`:

TestAuditParser class from the test_audit.py module:	
test_parse_xperms()	Test that correct access vectors are generated from a set of AVC denial messages.

5.1.3 Testing access.py Module

In this module, classes `AccessVector` and `AccessVectorSet` were extended.

Testing `AccessVector.__init__()`:

TestAccessVector class from the test_access.py module:	
test_init()	Test that AccessVector.xperms is a dictionary.

Testing `AccessVector.merge()`: this method must correctly merge permissions and extended permissions of two access vectors.

TestAccessVector class from the test_access.py module:	
test_merge_noxperm()	Test merging two AVs without extended permissions.
test_merge_xperm1()	Test merging AV that contains extended permissions with AV that does not.
test_merge_xperm2()	Test merging AV that does not contain extended permissions with AV that does.
test_merge_xperm_diff_op()	Test merging two AVs both containing extended permissions, but with different operations.
test_merge_xperm_same_op()	Test merging two AVs both containing extended permissions with the same operation.

Testing `AccessVector.add_av()`: this method adds access vector to the set.

TestAccessVectorSet class from the test_access.py module:	
test_add_av_first()	Test adding first access vector to the access vector set.
test_add_av_second()	Test adding second AV to the set with same source and target context and class.
test_add_av_with_msg()	Test adding audit message.

Testing `AccessVector.add()`: this method just creates an instance of `AccessVector` class and passes the AV to the `AccessVector.add_av()` method.

TestAccessVectorSet class from the test_access.py module:	
test_add()	Test adding access vector to the set.

5.1.4 Testing policygen.py Module

In this module, `PolicyGenerator` was extended to generate extended permission access vector rules.

Testing `PolicyGenerator.__init__()`:

`TestPolicyGenerator` class from the `test_policygen.py` module:	
`test_init()`	Test that extended permission AV rules are not generated by default.

Testing `PolicyGenerator.set_gen_xperms()`:

`TestPolicyGenerator` class from the `test_policygen.py` module:	
`test_set_gen_xperms()`	Test turning on and off generating of extended permission AV rules.

Testing `PolicyGenerator.add_access()`:

`TestPolicyGenerator` class from the `test_policygen.py` module:	
`test_av_rules()`	Test that new implementation of the `add_access()` method does not break generating of standard AV rules.
`test_ext_av_rules()`	Test that correct extended permission AV rules are generated from access vectors.

5.1.5 Testing refpolicy.py Module

The `XpermSet` and `AVExtRule` classes were added to represent extended permission access vector rules.

Testing `XpermSet` class: this class represents extended permission values. The `add()` method adds values or ranges of values, the `extend()` method combines two `XpermSet` objects, and the `to_string()` method prints the rule.

`TestXpermSet` class from the `test_refpolicy.py` module:	
`test_init()`	Test that all atttributes are correctly initialized.
`test_normalize_ranges()`	Test that ranges that are overlapping or neighboring are correctly merged into one range.
`test_add()`	Test adding new values or ranges to the set.
`test_extend()`	Test adding ranges from another `XpermSet` object.
`test_to_string()`	Test printing the values to a string.

Testing `AVExtRule` class: this class is similar to the `AVRule` class. The `from_av()` method creates the rule from an access vector and the `to_string()` method prints the rule to a string.

TestAVExtRule class from the test_refpolicy.py module:	
test_init()	Test that all attributes are correctly initialized.
test_rule_type_str()	Test printing the rule type.
test_from_av()	Test creating the rule from an access vector.
test_from_av_self()	Test creating the rule from an access vector that has same source and target context.
test_to_string()	Test printing the rule.

5.1.6 Integration Tests of Extended Permissions

Integration tests were written to check audit2allow functionality in real world situation. First, SELinux policy module with extended permission AV rules is loaded. Testing program then tries to call ioctl on a file with different parameters. AVC denials are collected and sent to audit2allow with different command-line options.

5.2 Testing Implementation of Mislabeled Files

New unit tests were written to cover new functionality.

5.2.1 Testing audit.py Module

In this module, the `AVCMessage.from_split_string()` method was extended to parse path field. Test cases:

TestAVCMessage class from the test_audit.py module:	
test_path()	Test that the path field is parsed.

5.2.2 Testing policygen Module

In this module, new configuration option was added to the `PolicyGenerator`. For the purposes of testing, `selinux.matchpathcon()` function used for checking the default context was replaced.

TestPolicyGenerator class from the test_policygen.py module:	
test_check_mislabeled_nothing()	Test no mislabeled files.
test_check_mislabeled_one()	Test one mislabeled file.

Chapter 6

Conclusion

Several situations where audit2allow provides too permissive or insecure solutions were identified. Extended permission access vector rules that provide more granular control were not supported by audit2allow. The audit2allow utility provided solutions that assumed that the context of object is correct.

Improvement of audit2allow were implemented.

The audit2allow can be further improved to detect situations where the correct solution is to use different tool.

Bibliography

[1] AppArmor. *Ubuntu Wiki*. [Online; accessed 14-March-2018].
Retrieved from: https://wiki.ubuntu.com/AppArmor

[2] Contributors to SELinux. *NSA.gov*. [Online; accessed 14-March-2018].
Retrieved from:
https://www.nsa.gov/what-we-do/research/selinux/contributors.shtml

[3] Extended Permission Access Vector Rules. *SELinux Project Wiki*. [Online; accessed 28-March-2018].
Retrieved from: https://selinuxproject.org/page/XpermRules

[4] HowTos/SELinux. *CentOS Wiki*. [Online; accessed 27-March-2018].
Retrieved from: https://wiki.centos.org/HowTos/SELinux

[5] Introducing AppArmor. *SUSE Documentation*. [Online; accessed 14-March-2018].
Retrieved from: https://www.suse.com/documentation/sles11/book_security/data/pre_apparm.html

[6] Linux Security Module Usage. *The Linux Kernel Documentation*. [Online; accessed 14-March-2018].
Retrieved from:
https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/index.html

[7] Security-Enhanced Linux in Android. *Android Open Source Project*. [Online; accessed 14-March-2018].
Retrieved from: https://source.android.com/security/selinux/

[8] Security Guide. *Red Hat Customer Portal*. [Online; accessed 20-March-2018].
Retrieved from: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing

[9] SELinux User's and Administrator's Guide. *Fedora Documentation*. [Online; accessed 14-March-2018].
Retrieved from: https://docs-old.fedoraproject.org/en-US/Fedora/25/html/SELinux_Users_and_Administrators_Guide/index.html

[10] SELinux User's and Administrator's Guide. *Red Hat Customer Portal*. [Online; accessed 14-March-2018].
Retrieved from:
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index

[11] *Trusted Computer System Evaluation Criteria*. 1985.

[12] *ioctl(2) Linux Programmer's Manual*. 2017.

[13] *xattr(7) Linux Programmer's Manual*. 2017.

[14] Haines, R.: *The SELinux Notebook*. 2014. [Online; accessed 14-March-2018]. Retrieved from: http://freecomputerbooks.com/books/The_SELinux_Notebook-4th_Edition.pdf

[15] Hallisey, R.: Improvements to audit2allow. [Online; accessed 28-March-2018]. Retrieved from: https://github.com/fedora-selinux/selinux/pull/1

[16] Stoep, J. V.: *[PATCH 2/2 v6] selinux: extended permissions for ioctls*. selinux@tycho.nsa.gov (Mailing list). June 2015. [Online; accessed 28-March-2018]. Retrieved from: https://marc.info/?l=selinux&m=143412575302369&w=2

Appendix A

Contents of Enclosed Memory Media

TODO