BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SELINUX POLICY ANALYSIS TOOL

NÁSTROJ PRO ANALÝZU BEZPEČNOSTNÍCH

POLITIK V SELINUX

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR VÍT MOJŽÍŠ
AUTOR PRÁCE

SUPERVISOR Prof. Ing. TOMÁŠ VOJNAR, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2016

Master Thesis Specification/18601/2015/xmojzi05

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems Academic year 2015/2016

Master Thesis Specification

For: **Mojžíš Vít, Bc.**
Branch of study: Intelligent Systems
Title: **SELinux Policy Analysis Tool**
Category: Security

Instructions for project work:

1. Get acquainted with Security Enhanced Linux (SELinux).
2. Design SELinux policy analysis tool capable of (a) representing SELinux policies as well as given integrity goals, (b) identifying conflicts between them (i.e., capable of analysis of interactions between policy modules), and (c) providing information necessary for resolution of such conflicts.
3. Implement the tool so that it can be integrated with current SELinux user-space tools.
4. Demonstrate functionality of the developed tool on a non-trivial use case.
5. Evaluate the obtained results and discuss possible future improvements of the developed tool.

Basic references:

- Haines, R.: The SELinux Notebook, 2014. http://freecomputerbooks.com
- Mayer, F., MacMillan, K.,Caplan, D.: SELinux by Example: Using Security Enhanced Linux, Prentice Hall PTR, 2006.
- Khedker, U.P., Sanyal, A., Karkare, B.: Data Flow Analysis: Theory and Practice, CRC Press, 2009.
- Chess, B., West, J.: Secure Programming with Static Analysis. Upper Saddle River: Addison-Wesley, 2007.

Requirements for the semestral defense:
The first item and significant progress on the second item.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

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

L.S.

Petr Hanáček
Associate Professor and Head of Department

Abstract

This thesis deals with *mandatory access control* (MAC)-based security module policy analysis, focusing on SELinux. Because of insufficient capabilities of available analysis tools, new tool was designed and implemented with the needs of Red Hat SELinux team in mind. Its main uses will be as aid in policy development and support in SELinux usefulness evaluation. If the tool proves useful, it will be incorporated into *SELinux userspace tools* package SETools 4.

Abstrakt

Tato práce se zabývá analýzou politik pro bezpečnostní moduly založené na *mandatorním řízení přístupu* (MAC), se zaměřením na SELinux. Vzhledem k omezeným schopnostem dostupných nástrojů byl navržen a implementován nový nástroj. Jeho hlavní cíle jsou usnadnění vývoje bezpečnostních politik a pomoc při odhadu role SELinuxu v zabezpečení systému. V případě úspěšného nasazení bude nový nástroj začleněn do balíčku nástrojů SETools 4.

Keywords

SELinux, analysis, security policy, mandatory access control.

Klíčová slova

SELinux, analýza, bezpečnostní politika, mandatorní řízení přístupu.

Reference

MOJŽÍŠ, Vít. *SELinux Policy Analysis Tool.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

SELinux Policy Analysis Tool

Declaration

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Prof. Ing Tomáše Vojnara, Ph.D., jakožto akademického vedoucího, a Ing. Miroslava Grepla, jakožto technického vedoucího. Další informace mi poskytli Bc. Lukáš Vrabec a Ing. Petr Lautrbach. Uvedl jsem všechny literární prameny a pulbikace, ze kterých jsem čerpal.

. . . .
Vít Mojžíš
May 25, 2016

Contents

Chapter 1

Introduction

In computing systems, security policies are means to precisely specify desired (and therefore allowed) access to system resources. Systems where security is top priority, such as those used by military or government, can make use of formal security models that are designed and proven to deal with specific types of threats. Security policies, as well as the rest of software equipment in such systems, unconditionally adhere to exact set of rules defined by chosen model.

Creating policies for other environments, like enterprise systems or even personal computers, proves significantly more challenging because the goal is to increase security while not limiting normal activity of the system (no significant changes to the system). Instead of complete formal models, policy writers are guided by general security goals and security becomes only best effort endeavour. Among the most common security goals are, limiting information flow to and from protected resources, resource isolation and separation of duty. Policies , however, are written in low level languages to allow for fine grained control, which makes achieving such goals challenging. Encapsulation and interdependence of policy rules further complicates the task by clouding exact effects (reach) of policy statements. SELinux policy files are clear example, since single policy statement can affect resources throughout the whole system.

SELinux is focused on preventing unauthorised access to systems resources and mitigating effects of exploited application flaws by exercising principle of *least privilege*[1]. In current default SELinux policy for Fedora (*targeted*), privileged processes (mostly daemons run by the system) and highly used programs are assigned individual SELinux domains and are allowed (only) the access required for their normal functions. This means that, in case some process gets compromised, attacker is limited to domain of given process, which in most cases prevents *privilege escalation*. But what is the minimum set of privileges necessary for given application? Policy writers are often left to answer this question themselves by studying documentation and access attempts from several test runs of the application, since its developers are unaware of, or reluctant to learn SELinux. Ultimately, applications are usually allowed more access than necessary due to assumed requirements, untraced application changes or misused (or overreaching) policy rules. Making application policy too "tight" shows as application failure and is, in the worst case, brought back to policy writers attention by bug reports. Too "loose" policy, however, doesn't show unless exploited.

[1]SELinux supports *multi level security* as well, but that functionality is commonly not used and therefore not considered in this thesis.

This work explores currently available tools for *mandatory access control* policy analysis with particular focus on SELinux and describes a new tool developed in cooperation with Redhat SELinux team. The tool was designed to help increase the quality of SELinux policy through identifying the following.

- Reach (Access allowed by SELinux) of specified domain or group of domains used by one service

- Possibly dangerous data pathways allowed by SELinux – eg. information flow between (groups of) domains and security-critical resources

This information will be useful in several areas. For example, estimating impact of successful attack on specific service, evaluating feasibility of exploiting *common vulnerabilities and exposures* (CVE) on systems protected by SELinux, or explaining policy effects to non SELinux aware audience (package developers, administrators, etc.).

The rest of the work is organised as follows. In chapter 2, basic security concepts used in this thesis are explained. Chapter 3 provides overview of SELinux security module. Chapter 4 deals with currently available policy analysis tools, as well as the working scheme for new tool. Chapter 5 describes development process of the new tool and explains the reasons behind important design decisions. Suggested use cases for the new tool are outlined in Chapter 6. Chapter 7 brings summary of the work.

Chapter 2

Security Concepts

All non-trivial software inherently contains flaws and even though there are ongoing efforts to increase code quality, it will probably remain flawed in the foreseeable future. Reasons may vary, but the fact remains that some people will always try to exploit these flaws. This chapter describes concepts designed to protect computer systems from intrusions, or mitigate their impact.

2.1 The Reference Monitor

One of the first steps to make multi-user systems more secure was the *reference monitor concept* from the so-called Anderson Report [1], created as part of research led by the U.S. Department of Defence.

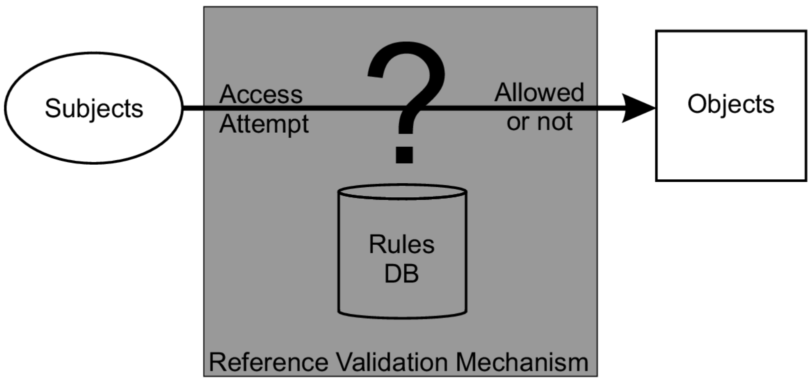


Figure 2.1: The reference monitor concept. [7]

> The function of the reference monitor is to validate all references (to programs, data, peripherals, etc.) made by programs in execution against those authorized for the subject (user, etc.). The Reference Monitor not only is responsible to assure that the references are authorized to shared resource objects, but also to assure that the reference is the right kind (i.e., read, or read and write, etc.).
>
> **A)** The reference validation mechanism must be tamper proof.
>
> **B)** The reference validation mechanism must always be invoked.

C) The reference validation mechanism must be small enough to be subject to analysis and tests to assure that it is correct.[1]

For the reference monitor to work, system has to isolate passive resources into distinct *objects* (eg. files) and active entities (running programs) into *subjects*. The *reference validation mechanism* then checks all access from subjects to objects against a security policy consisting of access control rules (see Figure 2.1). Security-related characteristics of subjects and objects are stored as *security attributes*. Examples of such characteristics are user identities for processes and access permission modes (owner-group-world) for files in Linux.

2.2 Discretionary Access Control (DAC)

Basic access control currently prevalent in most Linux distributions for personal computers is a form of DAC as defined in *The Orange Book* [10] (class C2). Access to objects is restricted based on the identity of users or groups to which they belong. The security policy rules are usually fixed leaving the security attributes as the only means of modifying access rights. The controls are discretionary in the sense, that authorized users are allowed to change the security attributes of objects, thereby specifying whether other users have access to them.

The idea behind DAC was to emulate real world security. However, unlike in real world, most of the actions we do in computer system are done by complex software not of our design. Instead of users, the access rights are effectively given to software run by users. Malicious, or flawed and exploited software is allowed the same access as the user running it, which leaves the security of the system on its users.

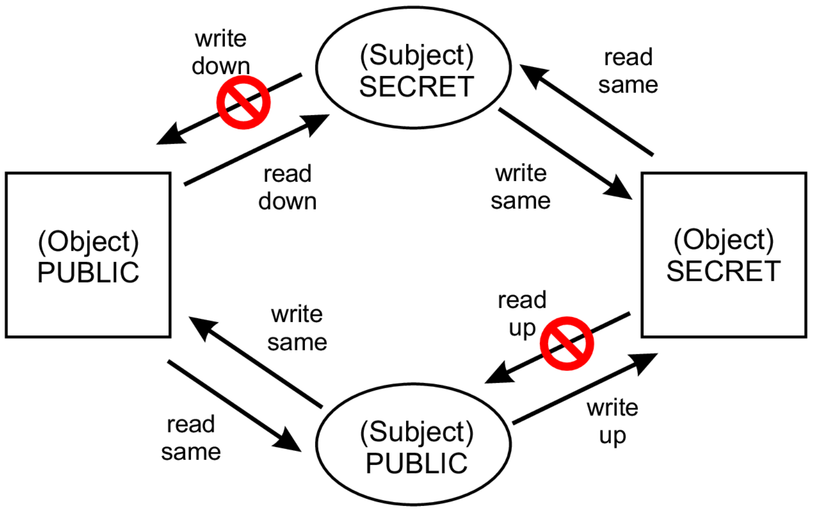


Figure 2.2: Simple MLS model with two security levels *Public* and *Secret* [7]

2.3 Mandatory Access Control (MAC)

To overcome issues of DAC, it was necessary to provide more fine-grained access control (process-level instead of user-level) and define policy base which would not be at the discretion of users or even system administrators (mandatory base). U. S. military provided significant part of funding for this work and added new focus which was protecting confidentiality of classified data. So far most common form of MAC adheres to *Multilevel Security* (MLS) concept based on *Bell-LaPadula* security model. In MLS, all objects are assigned a security level and all subjects a security clearance. Subjects can access only objects for which they are cleared and data can move only from lower levels to higher (no “read up”, no “write down”, as shown in Figure 2.2).

In non-military environment (notably PC), MLS has proven to be too restrictive, especially when confidentiality is a secondary objective. Instead, MAC for common systems has to be more flexible and focused on the malicious software issue, working as an extension of DAC. Currently there is several implementations of security modules for Linux based on MAC, most popular of which are AppArmor[1], SELinux[2] an SMACK[3]. All three are implemented as modules for *Linux Security Modules* (LSM) framework and provide program-level access control. SELinux will be further discussed in chapter 3.

[1]http://wiki.apparmor.net/index.php/Main_Page

[2]Security Enhanced Linux http://selinuxproject.org/page/Main_Page

[3]Simplified Mandatory Access Control Kernel http://schaufler-ca.com

Chapter 3

SELinux

SELinux is an implementation of MAC security module currently used in several Linux distributions (Fedora, Ubuntu, Debian, Gentoo, . . .). Starting as *Flux Advanced Security Kernel* (FLASK) development by the Utah university Flux team as part of US Department of Defence contract, SELinux was later enhanced by the NSA and released as open source software. Currently there are many independent contributors, most important of which are NSA (main branch), Tresys Technology (Reference policy) and Red Hat (Fedora/RHEL customisation)[1]. Following overview is based on [4] and [7].

3.1 Structure

This section provides an overview of main SELinux components and their role within the system. Figure 3.1 shows a high level diagram of component interaction. Starting from the bottom:

a) The SELinux security server resides in the kernel, while the policy is loaded from userspace via functions in *libselinux* library, which allows for more complex policy management (dynamic loading of policy modules). Decisions of the security server are enforced by *object managers*. Object manager is a subsystem designed to create and manage given set of resources (e.g. file system, interprocess communication, windows or tables). There are two types of object managers based on their location:

kernel space – These object managers deal with kernel services (files, sockets, IPC, etc.), utilising hooks into SELinux subsystem provided by *Linux Security Modules* (LSM) framework. Single kernel *access vector cache* (AVC) service is used for caching security server responses to all kernel-based object managers in order to decrease response time of same requests in the future. Content of the AVC is invalidated upon any policy change in order to allow for access revocation[2].

userspace – These object managers are contained within *SELinux aware* applications and provide MAC over special resources such as windows, or database tables. Examples of SELinux aware applications and services are X-Windows, database services (PostgreSQL), or GNU/Linux passwd command. Such applications have to maintain their own AVC's (usually using *libselinux*), if AVC is to be used. Separate

[1]Open source community significantly contributes to overall development.

[2]SELinux does not fully implement access revocation. Access to some resources (memory mapped files, connection oriented sockets, . . .) is validated only upon first use.

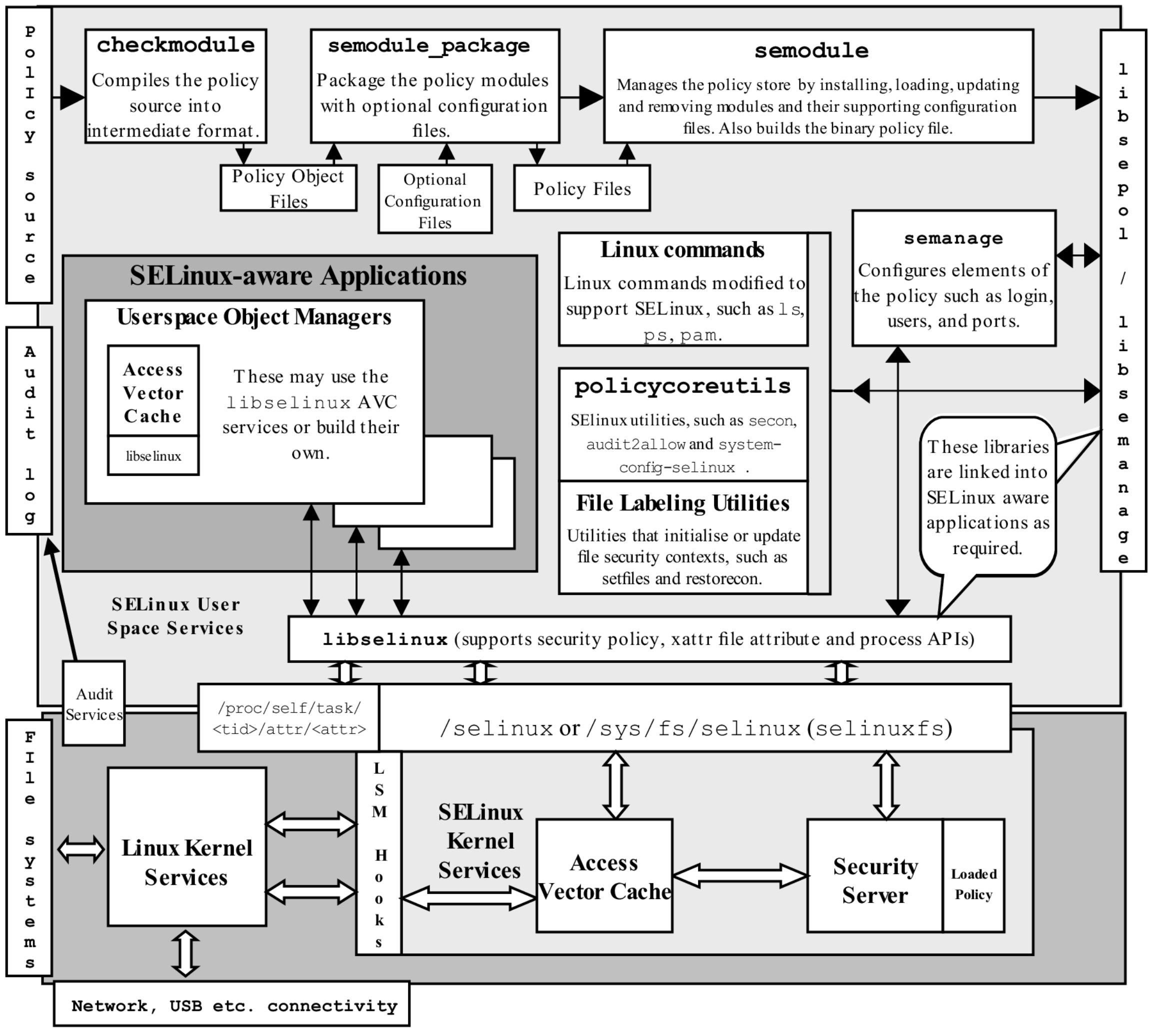

Policy source
checkmodule
Compiles the policy source into intermediate format.
semodule_package
Package the policy modules with optional configuration files.
semodule
Manages the policy store by installing, loading, updating and removing modules and their supporting configuration files. Also builds the binary policy file.
libsepol / libsemanage
Policy Object Files
Optional Configuration Files
Policy Files
SELinux-aware Applications
Userspace Object Managers
Access Vector Cache
libselinux
These may use the libselinux AVC services or build their own.
Linux commands
Linux commands modified to support SELinux, such as ls, ps, pam.
semanage
Configures elements of the policy such as login, users, and ports.
policycoreutils
SElinux utilities, such as secon, audit2allow and system-config-selinux .
File Labeling Utilities
Utilities that initialise or update file security contexts, such as setfiles and restorecon.
These libraries are linked into SELinux aware applications as required.
Audit log
SELinux User Space Services
libselinux (supports security policy, xattr file attribute and process APIs)
Audit Services
/proc/self/task/<tid>/attr/<attr>
/selinux or /sys/fs/selinux (selinuxfs)
File systems
Linux Kernel Services
LSM Hooks
SELinux Kernel Services
Access Vector Cache
Security Server
Loaded Policy
Network, USB etc. connectivity

Figure 3.1: High Level SELinux Architecture - Showing the major supporting services.[4]

policy servers can be used instead of relying on SELinux kernel. Such policy servers can focus solely on resource classes provided by given object manager, significantly simplifying policy writing.

b) SELinux policy can be monolithic, but more commonly consists of a base policy containing mandatory data (object classes, permissions, etc.) and additional smaller modules usually supporting single service/application. All policy modules are held in *policy store* (managed by *semodule*) in order to be available for conversion to binary format that can be loaded into the security server (*libsepol/libsemanage*).

c) Policy source (top left corner) can be supplied in following three ways:

- *SELinux policy language* source code with m4 macro support. Suitable for smaller policies such as SE for Android.
- Using high level macros defined in the *reference policy* that significantly simplify writing more complex and structured policies.

- Using *Common Intermediate Language* (CIL), recently added to SELinux framework. CIL is currently used as intermediate step in compilation of all SELinux policies.

Whole policy compilation process can be seen in the top portion of figure 3.1.

d) All security events are logged using audit services (even when SELinux is not enforcing access control). Log file location depends on which audit daemons are running (auditd/rsyslogd/setroubleshootd).

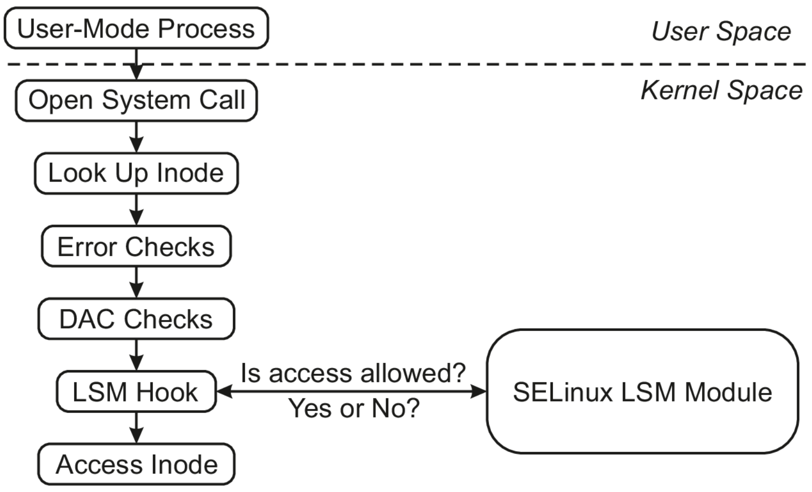


Figure 3.2: LSM hook architecture.[11]

3.1.1 LSM Framework

LSM Framework allows for security modules to be linked to the system kernel and further restrict default identity-based DAC security. Compatible 3rd party access control mechanisms can be registered to the framework and are then consulted for access validation of specific kernel calls (for list of affected kernel services, see table 3.1). This is done by set of *security function hooks* and additional data structures, residing in the kernel system call logic. These hooks take effect after the standard DAC access checks but before the resource is actually accessed by the kernel on behalf of the caller (see figure 3.2). As a result LSM is only used if standard access checks succeed and therefore there are no LSM logs of DAC denials. Each LSM hook can deal with multiple access permissions for one or more object

Program execution	Filesystem operations	Inode operations
File operations	Task operations	Netlink messaging
Unix domain networking	Socket operations	XFRM operations
Key Management operations	IPC operations	Memory Segments
Semaphores	Capability	Sysctl
Syslog	Audit	

Table 3.1: Kernel services controlled by LSM hooks.

classes. LSM framework by itself doesn't provide any security services. If no other security module is loaded, *capabilities* module is used, resulting in standard DAC access control. The SELinux LSM module can be seen in bottom part of Figure 3.1. As it was derived from the Flask architecture, it comprises of three parts: security server, object managers, and the access vector cache.

3.2 Type Enforcement

The core of SELinux functionality is *type enforcement*. It consists of labels assigned to each object (file, symbolic link, socket, pipe, . . .) and subject (process) in the system and a set of rules governing their interaction. The fact that access rules operate (via labels) on processes allows SELinux policy writers to control access based on the function and security properties of individual programs in addition to restrictions placed on users.

3.2.1 Security Context

The labels are known as *security context* and are kept in extended attributes of each object[3]. Security context is represented by variable-length string consisting of user, role, type and optionally security range as follows:

```
user:role:type[:range]   (system_u:object_r:passwd_file_t:s0)
```

user	SELinux user that is mapped to Linux user (n to n relation). Can be associated with roles, allowing the SELinux user to use them.
role	SELinx role (`object_r` by default for objects). Can be associated with types, allowing the SELinux user in given role to use them (limits SELinux user to policy rules containing listed types).
type	SELinux type is the main element in security context. Used in type enforcement rules.
range	Also known as *level* is only used in MCS or MLS modes[4]. It is made up of sensitivity (single level or range) and zero or more categories (e.g. s0:c10.c15, s0-s15:c0.c1023)

3.2.2 Types and Attributes

As expected, types are the most important part of type enforcement. Except for object classes, SELinux types are the only means of distinguishing between different resources and processes[5] (from SELinux policy point of view). The use of types in policy writing is made easier by *attributes* and *aliases*. *Attributes* are means of referring to a group of types by a single identifier. Policy language supports using attributes in place of type identifiers in most cases, including type enforcement rules. *Aliases* are used to define alternative names for a type. It is only a convenience mechanism and does not provide any additional benefits compared to using the original type identifier.

[3]In order to work properly, SELinux requires the filesystem to support extended attributes.
[5]Considering only on type enforcement.

3.2.3 Object Classes and Permissions

Each object in the system is instance of a single class that defines its purpose. Each class is associated with a set of permissions that describe what services can object of given class handle. Object classes therefore represent categories of objects and interfaces for interaction with them. SELinux works with kernel and userspace object lasses. The former represent all objects accessible via kernel calls (files, sockets, ...), and permissions associated with them are based on available LSM hooks. Userspace object classes represent resources available via userspace object managers (windows, tables, ...).

3.2.4 Type Enforcement Rules

Only *access vector* rules[6] will be mentioned, since they specify all access permissions relevant to this work. As of now, the policy language supports four types of AV rules:

`allow`	Translates into access allowed between two types.
`dontaudit`	Causes audit messages for specified access not to be generated.
`auditallow`	Causes audit messages to be generated even in case specified access was allowed.
`neverallow`	Translates into access that is never to be allowed. Any rule allowing it will cause failure of policy compilation.

Despite different purpose, all mentioned rule types have the same syntax. Each rule consists of five parts:

Rule type	`Allow`, `dontaudit`, `auditallow`, or `neverallow`.
Source type(s)	SELinux type receiving (being granted) access. Should be domain type.
Target type(s)	Type of object(s) to which access is being allowed. Can be both domain and resource type(s).
Object class(es)	Class(es) of target object(s).
Permission(s)	Set of access permissions being allowed. Also called the access vector. Has to be subset of permissions associated with specified classes.

By far the most common rules (and most important for this work) are of type `allow`, therefore further references to policy rules should be considered to mean `allow` rules. `Neverallow` rules are mostly used in *base* policy module, guarding access to most restricted resources[7]. Special attributes are often used to create exceptions in such rules (never allow any type without given attribute to be assigned specified access). `Dontaudit` rules are useful in cases where process only tests availability of resources, not intending to use them (common in network related daemons). `Auditallow` rules are used almost exclusively for testing/debugging.

[6]The resulting policy data (permissions to given object class) are stored in the form of bit mask (access vector).

[7]It should be noted that `neverallow` rules can only prevent direct assignment of specified permissions, which severely limits their application.

3.2.5 Process Domain Transition

Domain transitions are essential feature of type enforcement, since they facilitate change of process type (domain) upon file execution[8]. This allows processes to be confined in their own domains. Process running in domain `A` can transition to domain `B` by execution of file (entrypoint) labelled `C` when the following permissions are in place:

```
allow A C: file {getattr execute};
allow B C: file entrypoint;
allow A B: process transition;
```

[8]Domain transitions can (when specifically allowed to) be triggered without file execution. But that is very rare occurrence (reserved for parts of SELinux itself and system kernel).

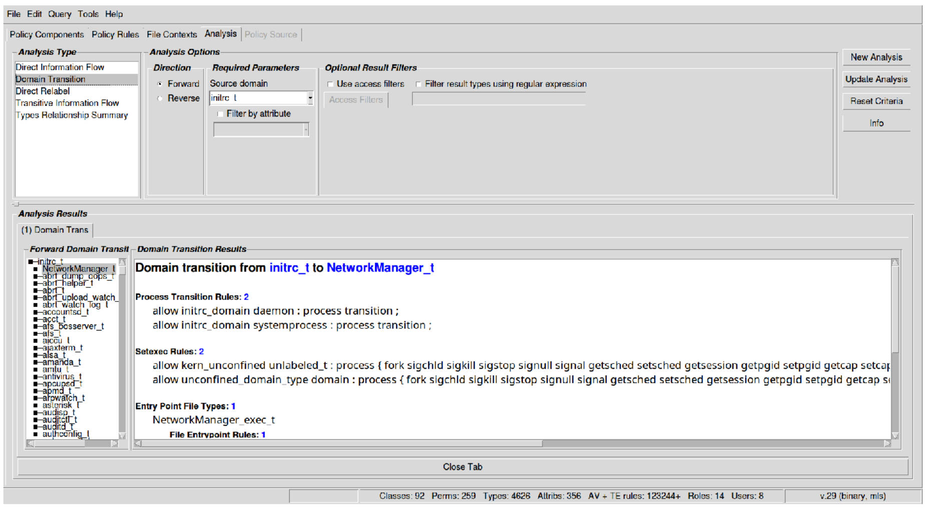
Chapter 4

Current State of SELinux Policy Analysis

As of few recent years, most operating systems started to integrate some form of mandatory access control. As mentioned in the introduction, managing security policies for such security modules can, with growing policy size, surpass abilities of policy writers. Policy analysis offers tools to locate inconsistencies and evaluate attributes of the policy as a whole, as opposed to individual policy files.

4.1 Existing Analysis Tools

Several analysis tools were developed to assist in successful policy development. This section describes their capabilities in context of SELinux policy. Unfortunately, to the best of authors knowledge, none of these tools are publicly available, except for Apol.



Analysis Type: Domain Transition; Direction: Forward; Source domain: initrc_t

```
initrc_t
  NetworkManager_t
  abrt_dump_oops_t
  abrt_helper_t
  abrt_t
  abrt_upload_watch_
  abrt_watch_log_t
  accountsd_t
  acct_t
  afs_bosserver_t
  als_t
  aiccu_t
  ajaxterm_t
  alsa_t
  amanda_t
  amtu_t
  antivirus_t
  apcupsd_t
  apmd_t
  arpwatch_t
  asterisk_t
  audisp_t
  auditctl_t
  auditd_t
  authconfig_t
```

```
Domain transition from initrc_t to NetworkManager_t

Process Transition Rules: 2
    allow initrc_domain daemon : process transition ;
    allow initrc_domain systemprocess : process transition ;

Setexec Rules: 2
    allow kern_unconfined unlabeled_t : process { fork sigchld sigkill sigstop signull signal getsched setsched getsession getpgid setpgid getcap setcap
    allow unconfined_domain_type domain : process { fork sigchld sigkill sigstop signull signal getsched setsched getsession getpgid setpgid getcap se

Entry Point File Types: 1
    NetworkManager_exec_t
        File Entrypoint Rules: 1
```

```
Classes: 92   Perms: 259   Types: 4626   Attribs: 356   AV + TE rules: 123244+   Roles: 14   Users: 8      v.29 (binary, mls)
```

Figure 4.1: Apol user interface, showing results of domain transition query.

4.1.1 Apol

Apol is a graphical tool developed by Tresys Technology as part of SETools Policy Analysis Suite[1]. Analysed policy can be in form of source, monolithic binary, or modular binary files. Primary use is examining, searching, and relating policy components (types, object classes, etc.), and policy rules (allow, type transition, etc.). Some more complex queries are supported, including domain transitions, information flows, and relabelling permissions (only single source/destination domain type is supported). The user needs to be well versed in policy writing as it was developed for policy administrators.[8]

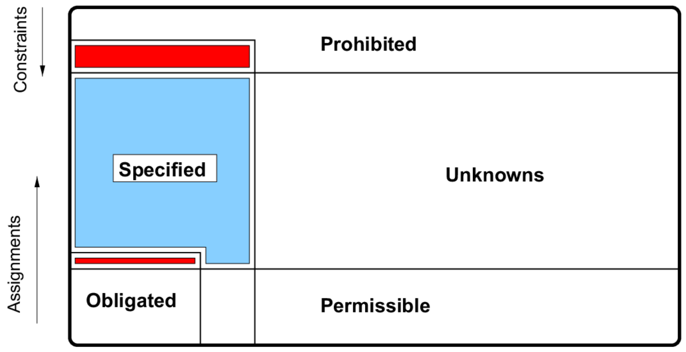


Figure 4.2: A realistic access control space: the specified space conflicts with the prohibited space and the unknown space.

4.1.2 Gokyo

Jaeger et al. [6] developed tool capable of identifying and resolving conflicting policy specifications. Together with security policy, safety requirements in the form of constraints are specified. Gokyo implements approach called *access control spaces*, dividing active policy to three permission sets:

- Permissions assigned to a subject type.
- Permissions precluded from a subject type by a constraint.
- Permissions whose assignment or preclusion status is unknown.

Relations between those sets are then evaluated (see Figure 4.2). Permissions that are both assigned and precluded form integrity conflicts, are further processed by the tool in order to estimate optimal resolution. Similarly permissions that are neither allowed nor explicitly prohibited might be interesting for policy writers[2]. Example graphical output can be seen in figure 4.3.

4.1.3 Security-Enhanced Linux Analysis Tool (SLAT)

Gutteman et al. [2], [3], defined information flow model and implemented it in tool named SLAT. Flow transition (transitive property) represents a possibility of information transfer – write operation transfers information from process to resource and read operation transfers

[1]https://github.com/TresysTechnology/setools

[2]This case cannot occur in SELinux since it block any access that is not explicitly allowed.

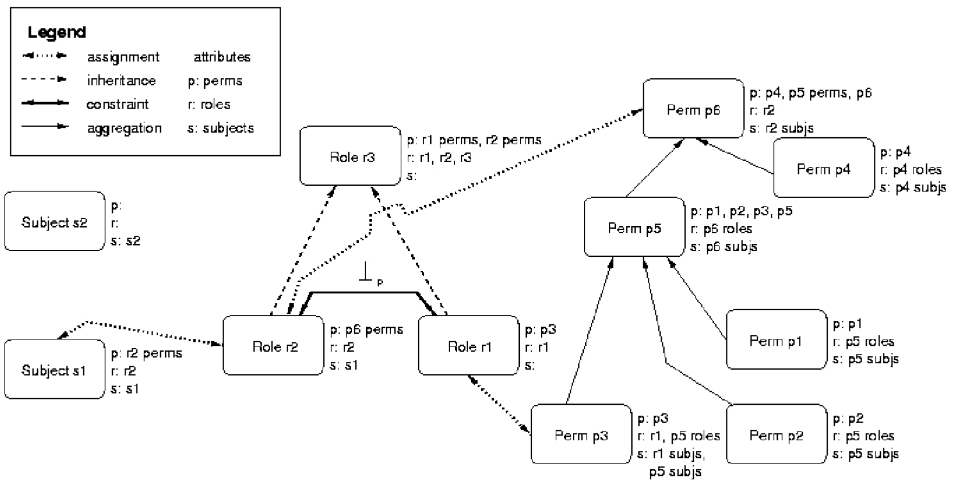


Figure 4.3: Example access control representation by gokyo tool (the fields p:, r:, s: refer to the permissions, subject types, and subjects assigned to these entities, respectively). [5]

information from resource to process. SELinux enforcement model together with security policy were represented using formal language based on linear temporal logic. Simple logic program queries were then used to analyze the policy. The flow model was later used by Sarna-Sota et al. [9] to implement PAL (Policy Analysis using Logic Programming).

4.1.4 Policy Visualization Analysis (PVA)

As the name suggests, PVA is focused on visual presentation of policy information. The tool supports *semantic substrates* and *adjacency matrix* layouts for visualization as well as query-by-example based visual query formulation. User study suggests that first-time users were able to use PVA significantly more efficiently than Apol. As of query functions, PVA offers the most complex queries, compared to the rest of listed tools. The framework allows for specifying system TCB and domain groups which can then be used in user queries.[12]

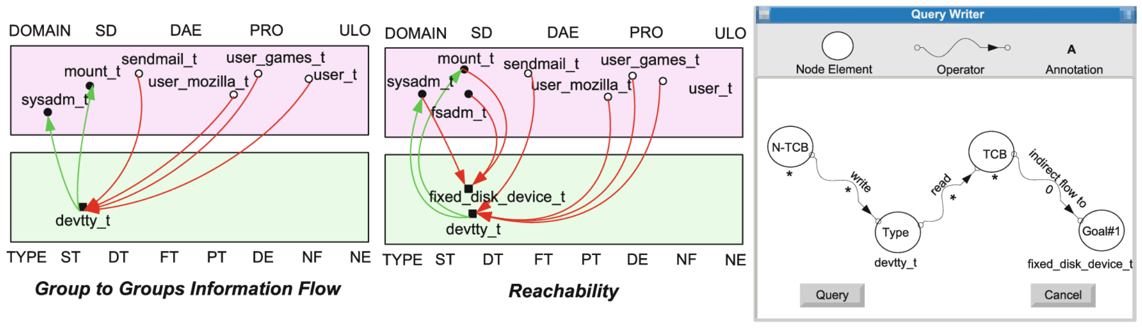


Figure 4.4: PVA user interface. Results of two queries and visual query formulation (right).

4.2 New Analysis Tool Proposition/concept

Because of limited analysis capabilities of apol (no domain grouping or TCB awareness, queries limited to single source/destination domain) and unavailability of any other analysis tool, Red Hat SELinux team initiated new project aimed at creating new tool. This tool will have two main purposes:

- Aid in policy development – detecting inconsistencies with security goals, regression testing
- SELinux usefulness evaluation – limiting impact of successful attack (CVE examples)

To satisfy these requirements, I propose the following working scheme[3]:

1. Determine System TCB (trusted computing base) equivalent.
2. Automatically determine "computing base" for each service.
3. Identify information paths between individual "computing bases" (most importantly those from/to System TCB) - query phase.

4.2.1 System TCB

> The heart of a trusted computer system is the Trusted Computing Base (TCB) which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based.[10]

The TCB will be selected from types belonging to *kernel* and *system* categories, together with types used by SELinux module itself. Most of this step will be done manually, but the result will be saved as static resource and there will be no need to update it until significant changes occur in target system kernel, or SELinux module. The selection process will take place in the last project phase, as some of the tools query functions will be necessary. Work of Jaeger et al. [6] will probably be used as reference point.

4.2.2 Individual Service "Computing Bases" (type groups)

This step will provide disjunct sets of domains (and corresponding resource types) concerning each SELinux enabled service/application based on user-specified criteria. User will decide upon types of pathways used in data flow analysis and sensitivity of the grouping mechanism (merging threshold). This parametrisation should influence granularity of the resulting domain groups (some services might be grouped together or divided into smaller segments based on their security policy). Default setting should yield grouping corresponding to policy files, where each domain was defined. Name-based scan may be used to provide initial grouping. Concept of resulting simplified policy graph can be seen in figure 4.5. "Type groups" and "domain groups" will both be used throughout this document to describe the same set of objects. Term "domain groups" should empathise subject types contained in the group (e.g. when discussing access granted to given group – object types have no access since they represent passive resources).

[3]This scheme is partially based upon work of professor Xu et al. [12] and Jaeger et al. [6].

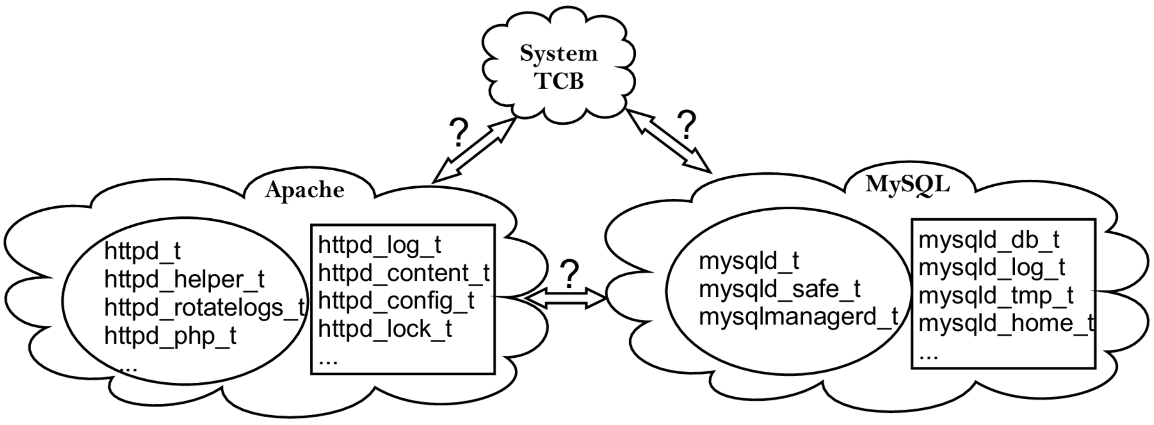


Figure 4.5: Concept of simplified policy graph generated by the second step (domain grouping).

4.2.3 Queries

The last step will entail user queries over the structure created so far. Grouping from the last step will have significantly decreased number of vertices in the resulting policy graph, hopefully resulting in reasonable query times (at least for basic queries). Available queries will range from listing neighbouring nodes of specified service, to deep scans of the whole policy graph in search for potentially dangerous information paths (eg. to system TCB). Queries concerning single service/application could be especially useful for users who are not well versed in SELinux policy writing, because they will show relationships between services, or services and resources, rather than SELinux types. More complex scans, on the other hand, will be targeted on policy writers since they will assist in evaluating overall security, or impact of more significant changes in policy. Query output will be mainly text, in accord with other utilities in SETools package. Optionally, simple queries will be visualised as graphs using NetworkX framework[4]. Policy data will be drawn from currently loaded system policy using SETools v4 package[5].

[4]NetwokX github repository https://networkx.github.io/

[5]Tresys Technology SETools repository https://github.com/TresysTechnology/setools

Chapter 5

Design and Implementation

5.1 Analysis Focus

SELinux is a complex security system with wide range of applications from servers all the way to mobile devices. Actual security level depends on the application since it is usually a compromise between ease of use of the host system and its ability to resist possible attacks.

This work focuses on best effort security which is most widely used and has the most room for improvement regarding security hardening. Primary target platforms are Fedora and RHEL (Red Hat Enterprise Linux), but future versions should support wider range of Linux distributions and possibly even Android OS.

5.1.1 Examined SELinux Rules

Both Fedora and RHEL are by default using SELinux in *Targeted* mode, which relies almost solely on type enforcement[1]. Future references to permissions or any form of access are meant only with respect to type enforcement (disregarding MLS rules).

5.1.2 Limitations

SELinux has several supporting features that allow for more fine grained control based on system configuration. For example SELinux *users* and *roles* can be used to limit domain transitions to a subset of existing domains, further tightening system security.

This work disregards such features, because they behave differently between systems and can't be evaluated statically (.i.e without the use of live target system or specific system configuration that could be used for simulation). Even without mentioned features, existence of given permission in SELinux policy does not guarantee feasibility of executing corresponding action. As a result, analysis described in this work considers only the worst case scenarios[2] and should be treated as such.

5.2 Type Grouping Development

At this point final SELinux policy (i.e. binary loaded into SELinux kernel) doesn't contain any information about policy modules from which individual rules and types originated.

[1]With the exception of virtualization and container support, both of which use some MLS features.

[2]With few exceptions that are controlled by the user, such as conditional policy statements.

Parsing policy files to gather corresponding types would require complex M4 macro analysis since type names often aren't assembled until first steps of policy compilation (*checkmodule*).

First attempts to form type groups involved mostly name based scanning of available file types. This resulted in relatively accurate approximation of policy modules, except for a few cases where the naming conventions were not followed properly or a service name was a substring of another (e.g. rpc, rpcbind; su, sudo). Reliance on on specific naming conventions would probably later cause portability issues.

Several attempts were made to utilise data flow analysis, but the resulting groups did not match services and their resources. Even after excluding rules originating in attributes, there are instances where a domain has more access to given resource than the primary consumer (domain corresponding to the same service as given resource). Because of this, dynamic type grouping was abandoned until its need arises during analysis process.

5.2.1 Common Intermediate Language

During implementation of this tool, SELinux in Fedora followed upstream example and introduced new step in compilation involving *CIL* (Common Intermediate Language). Policy files are now available in two different policy languages, where CIL files contain fully expanded type names, while still keeping modules separate. This also provides more reliable source for type grouping since CIL files reflect changes in policy caused by dynamic module loading and are available without any additional package or repository as is the case of "high level" policy files.

Namespacing, which was introduced together with CIL, should in future releases provide module names for type and rule definitions from binary policy, hence remove the requirement of administrator rights necessary for gathering CIL files.

5.3 Trusted Computing Base Equivalent

The Orange Book [10] requires that the TCB implements the reference monitor concept, i.e. it:

1. is tamper resistant
2. cannot be bypassed
3. is correctly implemented

Assuming SELinux is correctly implemented, policy can only influence the first two conditions. The first one by not allowing any non-TCB domain write access to TCB types, which will be our main focus, and the second one by ensuring that all processes are confined in their respective domains.

As Fedora wasn't designed around strictly defined TCB, we can't expect that some part of the system will completely fit the TCB requirements. In fact, the second reference monitor condition cannot be fulfilled in Fedora because of *unconfined_domain_type* attribute (and other less permissive ones[3]) that effectively disable SELinux control over processes running in domains with said attribute.

At this point most programs executed directly by user obtain unconfined domain label[4], which means that third-party software often has virtually unlimited access to almost all

[3]For the complete list use `#seinfo -a | grep unconfined`

[4]Only in default system configuration.

data in the system. With such security breach in place, even perfectly constructed policy can be bypassed by misslabelling key files or by directly exploiting one of the unconfined domains. It is therefore vital that system administrators assign users *confined* SELinux user profile.

That being said, we can label security-related part of the system as trusted and work towards making it tamper resistant as it should be. Bypassing can be solved by gradually reducing permissions of unconfined domains[5], because most of the currently unconfined processes do not need access to security related parts of the system.

5.3.1 Selection of Security Related Types

The selection of security related types was inspired by RHEL-7 Common Criteria work, which includes minimal system build that still contains key security features. The first step was to map essential objects from the minimal system build to SELinux types. Both high level and CIL policy files were used in this step. CIL files contain fully expanded type names, attribute assignments and allow rules, while high level policy provides context in form of M4 macros and comments.

In order to speed up the process and make further adjustments easier, type grouping gathered from the CIL files was used. Meaning that the security related set can be specified using individual types or whole policy modules, where exceptions (i.e. types only marginally connected to given module, that aren't necessarily security related) can be excluded.

Next step was making sure that the set of selected types is as close to approximating TCB as possible. At this point only tamper resistance is in question because marking types as security related will not influence whether SELinux control can be bypassed by non-TCB processes. Therefore queries checking *write* and *append* access to security related resources (e.g. executables, configuration files, ...) were used[6]. Several iterations were necessary, since marking new type as security related can introduce new undesired access. Results are saved in `security_related.conf` and can be edited by the user (may be necessary for older versions of Fedora/RHEL).

As expected, there are several domains with write access to the resulting set. Most notably there are domains with write access to entrypoints (i.e. executables that can spawn a process running in given domain) to security related domains, namely:

```
abrt_t, abrt_dump_oops_t, abrt_handle_event_t, prelink_t, gluster_t,
pegasus_openlmi_logicalfile_t, rpm_script_t, rpm_t
```

None of these, except for Pegasus which is deprecated, can work properly without the access, but neither should they be treated as security related. This situation currently has not practical solution, other than creating special domains for each of listed services that would have the necessary permission, but would be entered only via secure entrypoints. Dynamic transition would not work in this example, since successful attack on the less privileged domains would result in gaining access to the new privileged domains as well.

[5]While providing the necessary access via trusted mediator, or other controlled means.

[6]At first using only temporary tool `TCB_checker.py` and later utilising the main tool interface.

5.4 Visualization

One of the biggest current problems of SELinux is its complexity. In order to understand even the most basic concepts and its implications towards operating system (e.g. security context, explicit access rules, confined and unconfined domains), user has to spend significant amount of time (at least a few hours) studying the topic. This fact discourages even people who come in direct contact with SELinux (system administrators, confined service developers/maintainers) and who could easily contribute. With growing number of confined services, policy writers are unable to keep track of changes that should be projected into SELinux policy.

In order to reduce this problem I decided to provide graphical representation of simple policy queries. This will allow the user to explore the surroundings (i.e. other services and resources it can access, or can be accessed by) of selected service or part thereof. Main target group of this feature will be confined service maintainers since their understanding of policy settings is crucial for increasing policy quality. They can point out unnecessary permissions or request specific access as the service evolves.

In order to simplify the output, I chose to limit the source or target of the query to single type or group of types represented by single node[7]. The user interface is based on *sesearch*, sharing some of its search parameters (e.g. selection of specific classes or permissions). Output example can be seen in figure 5.1

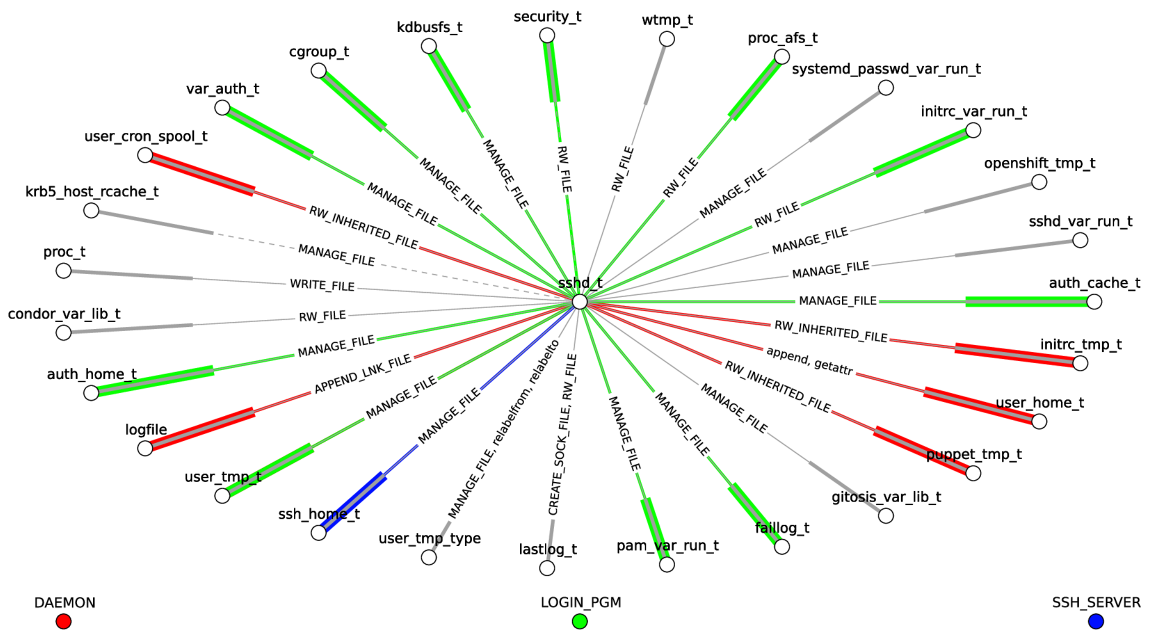


Figure 5.1: Visualization of write/append permissions granted to sshd_t domain with respect to files.[9]

[7]This limitation could be removed by utilizing more sophisticated graph visualization tool (ideally interactive).

[9]`#visual_query.py -s sshd_t -fb -c file -p write,append -fa domain`

5.4.1 Attributes

Attributes play significant role in policy specification, allowing for sharing of given set of rules. When dealing with only a subset of policy ,however, attributes can introduce permissions that seem to be (or actually are) out of place. For example all types corresponding to processes have to have attribute *domain*, which introduces 658 allow rules[10]. Permissions granted to attribute of selected source or target type are therefore distinguished by colours and can be filtered out using parameter "`-fa`" (filter attributes).

Attributes on the other side of allow rules are by default left untouched since they usually serve better at describing given rule and often correspond to large number of types. This behaviour can be changed by using parameter "`-ea`" (expand attributes), which will replace all attributes by corresponding types. Type names end by convention with "`_t`" which serves to distinguish them from other identifiers (in this case only attribute names).

5.4.2 Conditional Policy Statements

To deal with different system configurations, SELinux features set of user-configurable boolean variables that can enable or disable corresponding rules. Policy rules can be dependant on any boolean expression comprised of these variables (usually only single variable is used).

The tool allows for filtering of such rules based on given boolean settings. The worst case scenario is presented by default, meaning that all conditional rules are considered active. After specifying parameter "`-fb`" (filter booleans), boolean values given as argument are taken into account and the rest is loaded from active policy. In the graph representation, conditional rules are represented by dashed lines and corresponding boolean names are part of console output.

5.4.3 Permission Sets

The graph representation forced the aggregation of permissions from all selected classes to a single edge. Given how fined grained SELinux permission control is, it became necessary to group permissions together. Although not necessarily optimal in all cases[13], SELinux permission sets were used to fix this issue. To further distinguish permission sets form individual permission identifiers, sets are shown in capitals.

5.4.4 Type Grouping

As opposed to distinguishing iter-service and intra-service access, type grouping in visualization serves to further distance the user from SELinux inner workings. Whole groups of types can serve as nodes, representing services rather than separate types (see figure 5.2). Type grouping can be seen as higher level of abstraction. Obviously it is than often necessary to narrow down the search and/or filtering to decrease the amount of information that is to be displayed. If requested, attribute expansion takes place before type grouping. Unexpanded attribute names are shown in lower-case.

[10]Current targeted policy, Fedora 23. Complete list can be obtained by
`#sesearch -A -s domain -d` and `#sesearch -A -t domain -d`

[12]`#visual_query.py -s sshd_t -fb -dg -c file -p write,append -fa domain`

[13]Permission sets are often not disjunct, which may cause confusion if multiple rule classes are selected and only permission set for one of the classes is shown (permissions of other classes did not match any other permission set and were covered by the shown set).

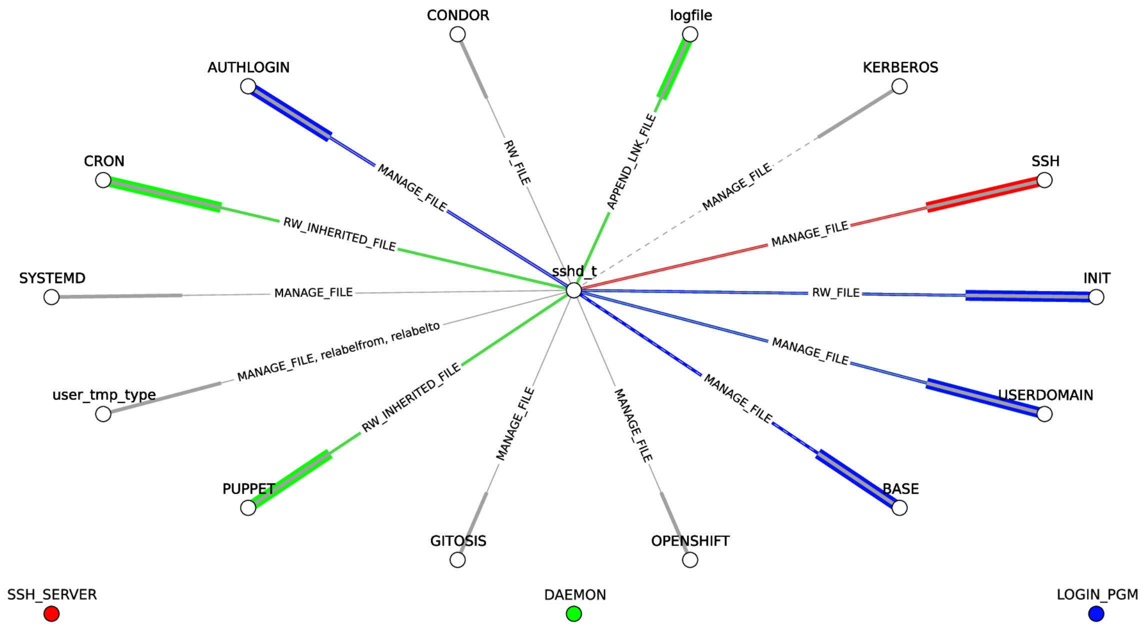


Figure 5.2: Visualization of write/append permissions granted to sshd_t domain with respect to files. Target types are grouped by policy module and access granted to *domain* attribute is omitted.[12]

5.5 Policy-wide Query

The main goal of this work is to take into account all policy rules at once as opposed to human per-module approach and find (potentially) dangerous pathways that are hard to locate for policy writers. In order to achieve this goal it was necessary to transform the policy into more suitable representation than that available to us via SELinux tools (i.e. set of rules allowing specific access). For this kind of search it is not important how was given access pathway created (via attribute or only to specific type, whether there are multiple rules allowing the same access, etc.), only that it was. Therefore graph representation was chosen.

5.5.1 Graph Construction

Allow rules are collected via SETools query interface from selected policy file, or from active policy if no file was specified. Depending on desired queries, subset of object classes can be specified in order to reduce size of the resulting graph. Collected rules are then transformed by the following steps.

- Boolean filtering – conditional statement of each rule is evaluated against given boolean setting
- Attribute expansion – rules concerning attributes are expanded to set of rules containing corresponding types
- Elimination of resource rules – rules giving access to resource types (usually via attribute) are removed

- Type grouping – optional replacement of types by corresponding type groups
- Rule grouping – forming of graph edges from rules with the same source and target

Resulting graph edges are represented by dictionaries of object classes (e.g. file, process, socket) containing sets of permissions. Sets were used for constant time of membership check. Complete graph is than saved to designated file. Duration of this procedure strongly depends on the amount of selected object classes and source policy size.

5.5.2 Graph Query

Graph_query is a modular tool that loads graph produced in previous step and executes user defined queries. It is designed to be used in two different scenarios. First one is system hardening, where selected queries are only applied on single policy graph, resulting policy issues are evaluated by policy writers and resolved if possible. More common use case should be regression testing, where selected set of queries is applied on two graphs generated from subsequent policy versions (parameter "`--diff POLICY_GRAPH2`"). The second policy graph (older version of policy) is used as baseline and only new issues are reported. This form of regression testing should be applied after each new policy build to ensure that new policy features or bug fixes do not have unforeseen consequences.

The difference between this approach and *SEDiff* tool is (except for significantly smaller resource requirements[14]) that *SEDiff* simply shows individual allow rules that are not contained in both given policies. However, such rules can have significant impact in context of the whole policy (completion of dangerous permission pathway), which is not shown.

Query Functions

New queries can be specified in `qraph_query_functions.py` using general search functions from `evaluation_functions.py` module. *Tuples*, *sets* or *lists* are expected as output from query functions. In order to be able to use other data types or customised text output, users can specify corresponding *diff* and *to_string* functions[15].

One of the most time consuming parts of this work was gathering information about useful queries. So far I was only able to formulate a few queries that are useful both for policy writers and testers. However I am confident that this will cease to be a problem once the tool is in use by larger user base.

Type Grouping

Type grouping during graph construction serves only to speed up the query process. First, given search function would be executed over type grouped graph, which is significantly smaller, and results would be verified (i.e. false-positives would be removed) on full graph. However, because of large number of false-positives and additional time required to load the grouped graph, resulting speed improvements were only marginal[16].

[14]SEDiff currently requires more than 12GB of RAM and several minutes to compare two recent Fedora policies.

[15]It is necessary to use given naming scheme in order for the tool to recognise new function purpose

[16]Tests were performed only on two query functions so far because query times are manageable even without this optimisation.

5.6 Future Development

It is very likely that development of this tool will continue. Development and gathering necessary information sparked interest in automated approach to policy analysis in both testers and developers, resulting in demand of features that are out of scope of the tools current working scheme.

5.6.1 Transitive Domain Reach

The closest goal is additional graph query interface for specific queries similar to *sesearch* that would take into account indirect access. Meaning access utilising other domains that can be manipulated by the source domain. Most of the code for determining complete domain reach is already in place. Covering both direct (i.e. there exists an entrypoint that can be written to and executed by the source domain, triggering transition to desired domain) and indirect manipulation (i.e. source domain only has write access to target domain entrypoint, meaning that execution of new code id delayed until exploited entrypoint is executed). Suitable form of relaying all the necessary information to the user is yet to be established[17].

5.6.2 SELinux Users and Roles

As seen in chapter 6, unconfined domains and generally domains with broad access across the system, can cause the current queries to produce false positives, or entries that cannot be influenced without significant structural change in policy. Some of these domains are not accessible for common users due to restricted SELinux users and roles that are commonly used by responsible administrators. Taking such restrictions into account (i.e. generating policy graph for specific SELinux user or set of roles) could increase usability of the tool.

5.6.3 Visualization

Utilising interactive graph visualization framework would allow for building significantly more complex policy graphs (suitable node positioning is problematic in current framework). Such graphs would be suitable for illustrating attribute expansion or as output of transitive access queries.

5.6.4 SEChecker

SECecker was a tool for simple static policy checking. It was designed to find poor policy writing practices, such as using single type for both domain and resources, unreachable domains, etc. Unfortunately it was abandoned (even though user base does exist) and eventually stopped working due to new SELinux features. Similar tests could be integrated into the graph builder interface.

[17]The output has to contain all the access as well as means of achieving it (i.e. via what domains, or files in case of delayed access)

5.6.5 Policy Loading

Policy data are currently loaded via SETools 4 interface which provides allow rules as specified in the policy source (after macro expansion). However, binary policy loaded into the kernel is in the form of hash table using source type, target type and object class as keys (much like the graph representation used in this work). Direct access to the binary policy could mean significant speed improvements of graph building process. Further research of this area could yield improvements in query times as well (kernel inspired optimisation).

Chapter 6

Use Cases

This chapter outlines suggested use cases for the tool and its interoperability with SETools. Given examples should be reproducible on any recent Fedora/RHEL release after installation of SETools 4 (see *readme* for complete installation guide). All the necessary data are available on the enclosed DVD.

6.1 Visualization

As explained in section 5.4, the visual representation of some policy sections is designed to promote interaction between package maintainers/developers and SELinux policy writers. It should make cooperation easier and give users insight into policy without the need to understand SELinux inner workings.

6.1.1 Bluetooth Daemon Example

Let us consider bluetooth daemon.

```
$ ps -axZ | grep bluetooth
system_u:system_r:bluetooth_t:s0 791 ? Ss /usr/libexec/bluetooth/bluetoothd
```

Shows its SELinux context `system_u:system_r:`**`bluetooth_t`**`:s0`, containing its domain type **`bluetooth_t`**. First we use type grouping to get a general idea of what service data can be accessed. To shift focus on access allowed just to bluetooth deamon, rules assigned to attributes *domain* and *daemon* are filtered out. See Figure 6.1.

```
$ ./visual_query.py -s bluetooth_t -c file -fa domain,daemon -dg
Boolean conditioned edges (dashed lines):

nis_enabled:
    SYSNETWORK
    NIS
kerberos_enabled:
    KERBEROS
authlogin_nsswitch_use_ldap:
    LDAP
    SYSNETWORK
    MISCFILES
```

The output was shortened since it contains a list of permission sets and corresponding permissions.

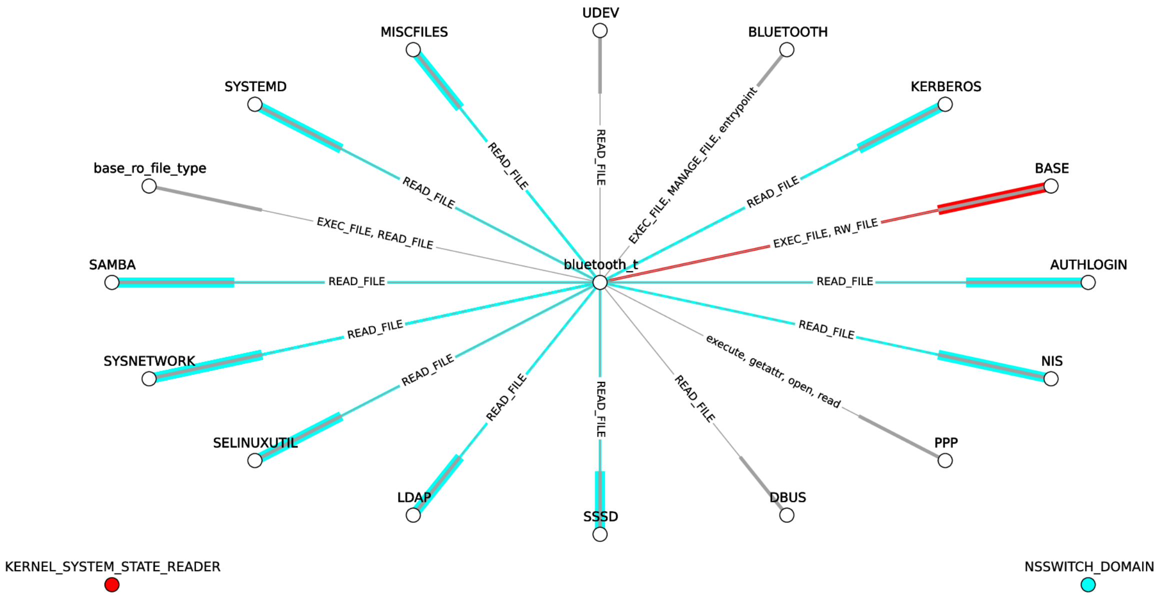


Figure 6.1: Visualization of permissions granted to bluetooth_t domain with respect to files. Target types are grouped by policy module and access granted to attributes *domain* and *daemon* is omitted.

Disabling type grouping gives us more detailed overview of permitted access (Figure 6.2).

```
$ ./visual_query.py -s bluetooth_t -c file -fa domain,daemon,nsswitch_domain
```

6.2 Graph Query

As explained in section 5.5, policy-wide queries are designed to find potentially dangerous pathways that are hard to locate for policy writers because they may have been created by undesired policy module interaction, M4 macro expansion, or attribute assignment.

6.2.1 Policy Graph

Only rules concerning *files* and *processes* will be used since currently implemented query functions do not require any other classes.

```
$ ./build_graph.py -fb -c file,process data/graph_3.13.1-158.16 -p
    policy_data/selinux-policy-targeted-3.13.1-158.16/policy.29
```

6.2.2 System Hardening

Let us first consider system hardening scenario. Query functions will be executed on latest Fedora 23 policy build. The output is shortened to first few entries per query function (full text is available on enclosed DVD in *examples*).

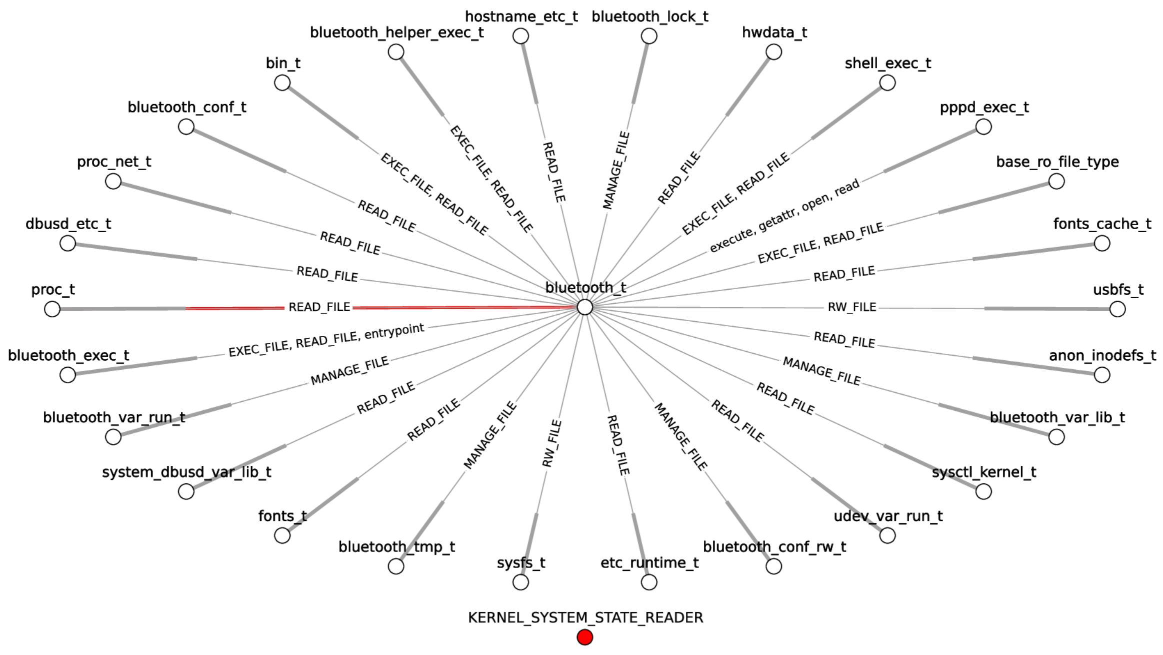


Figure 6.2: Visualization of permissions granted to bluetooth_t domain with respect to files. Access granted to attributes *domain*, *daemon* and *nsswitch_domain* is omitted.

```
$./graph_query.py data/graph_3.13.1-158.16 write_executable

write_executable:
   abrt_dump_oops_exec_t
      prelink_t, sysadm_t, pegasus_openlmi_logicalfile_t, rpm_t,
      systemd_tmpfiles_t, system_dbusd_t, abrt_dump_oops_t,
      abrt_handle_event_t, glusterd_t, kernel_t, rpm_script_t, mount_t
   abrt_etc_t
      puppetagent_t, rpm_t, sysadm_t, pegasus_openlmi_logicalfile_t,
      systemd_tmpfiles_t, abrt_t, system_dbusd_t, abrt_dump_oops_t,
      abrt_handle_event_t, glusterd_t, kernel_t, rpm_script_t, mount_t
```

`Write_executable` finds labels of executable files (from SELinux policy point of view[1].), that can be written to by some domain, and corresponding domains to which these entrypoints lead[2]. Output can easily be switched to show domains with given write permission, but showing both proved confusing. Even though it is very general (large percentage of the output does not present security risk), this query function will be especially important for regression testing.

What is interesting about this output is that file type designated for configuration files `abrt_etc_t` is an entrypoint to several domains. The reason is that there are some overreaching (unconfined) domains that can execute any file without domain transition.

[1]As explained before, files with given label might not have the DAC *executable* bit set
[2]Including domains which can execute the file without domain transition

```
$sesearch -A -t abrt_etc_t -p execute_no_trans -c file
   allow files_unconfined_type file_type : file { ioctl read write create
      getattr setattr lock relabelfrom relabelto append unlink link rename
      execute swapon quotaon mounton execute_no_trans open audit_access } ;
```

```
$./graph_query.py data/graph_3.13.1-158.16 transition_write

transition_write:
   abrt_dump_oops_t, abrt_helper_t, abrt_helper_exec_t
   abrt_handle_event_t, abrt_helper_t, abrt_helper_exec_t
   abrt_t, sendmail_t, postfix_postdrop_t
   auditadm_sudo_t, guest_t, user_home_t
   boinc_t, boinc_project_t, boinc_project_var_lib_t
```

This query produces triples *source_domain*,*target_domain*,*entrypoint* where *source_domain* can transition to *target_domain* by executing *entrypoint* to which it has write permission. Such access pathway should always present a red flag, since it effectively increases reach of source domain by access rights of target domain. Therefore it should also be a rare occurrence, usually "joining" domains inside single policy module, or showing power of unconfined domains, as seen in the example.

```
$./graph_query.py data/graph_3.13.1-158.16 write_to_security

write_to_security:
   iscsid_t
      systemd_passwd_var_run_t, sysfs_t
   piranha_pulse_t
      systemd_passwd_var_run_t, sysctl_rpc_t
   yppasswdd_t
      shadow_t, passwd_file_t
   slapd_t
      auth_cache_t, security_t
```

`Write/append_to_security` were created to promote isolation of security related types. Output lists non-security domains and security related types to which they have write access sorted by number of types that can be accessed from given domain.

Interesting error found during testing of this query is that `security_t` can be written to by large number of non-security domains. The reason is that it was assigned `non_security_file_type` attribute by mistake (the assignment was hidden inside a macro).

```
$sesearch -A -s glusterd_t -t security_t -c file -p write
   allow glusterd_t non_security_file_type : file { ioctl read write create
      getattr setattr lock append unlink link rename open } ;
```

```
$seinfo -tsecurity_t -x
   security_t
      boolean_type
      file_type
      filesystem_type
      mountpoint
      non_auth_file_type
      non_security_file_type
      base_typeattr_15
```

6.2.3 Regression Testing

Regression testing mode has the same output format and context except that entries found in both specified policy graphs are not shown. When executing on two subsequent policy versions, the output should contain small number of items and therefore should not have significant effect on policy maintenance/development process.

```
$./graph_query.py data/graph_3.13.1-158.16 write_executable,
   transition_write,write_to_security,append_to_security
      -d data/graph_3.13.1-158.15

write_executable:
   stunnel_log_t
      kernel_t, syslogd_t, mount_t, sysadm_t, abrt_dump_oops_t,
      abrt_handle_event_t, glusterd_t, rpm_script_t, logrotate_t,
      sandbox_t, pegasus_openlmi_logicalfile_t, system_dbusd_t,
      logadm_t, rpm_t, stunnel_t, systemd_tmpfiles_t
transition_write:

write_to_security:

append_to_security:
```

In this output we can see a new type introduced in policy version 3.13.1-158.16 and write access caused by attributes `logfile` and `non_security_file_type`, combined with unconfined domains.

```
$sesearch -A -t stunnel_log_t -p write -c file
   policy_data/selinux-policy-targeted-3.13.1-158.16/policy.29
Found 23 semantic av rules:
   allow files_unconfined_type file_type : file { ioctl read write create
      getattr setattr lock relabelfrom relabelto append unlink link rename
      execute swapon quotaon mounton execute_no_trans open audit_access } ;
   allow abrt_dump_oops_t non_security_file_type : file { ioctl read write
      create getattr setattr lock append unlink link rename open } ;
   allow syslogd_t logfile : file { ioctl read write create getattr setattr
      lock append unlink link rename open } ;
```

Chapter 7

Conclusion

This work explained why automated policy analysis is unavoidable in MAC based security systems and described the state of the art in SELinux policy analysis. Because of insufficient capabilities of available tools, new tool was designed and implemented. The tool was shown to be useful in both policy development and testing.

Necessary prerequisite for starting this work was thorough understanding of SELinux type enforcement and policy writing process. In order to achieve reasonable analysis speed it was first necessary to convert policy data into new representation, featuring efficient search of specific access pathways, and capable of storing all the required data. The most substantial challenge was gathering use case data from potential users and designing the tool interface to accommodate most important features. During implementation I had to compensate for changes in SETools 4 interface, which is still undergoing final stages of development.

It is expected that the tool development will continue outside of the scope of this diploma project. Future work entails extending the tool interface to incorporate more complex queries, refining query results (better filtering of false positives), and optimisations to increase overall performance (policy graph creation as well as query speed).

Bibliography

[1] James P. Anderson. *Computer Security technology planning study, volume II.* ESD-TR-73-51, Vol. II., Electronic Systems Division, Air Force Systems Command, 1972. Available at `http://csrc.nist.gov/publications/history/ande72.pdf`.

[2] Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdell. Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[3] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, January 2005.

[4] Richard Haines. *The SELinux Notebook [online].* 4. edition, 2014 [cit. 2015-11-13]. Available at `http://freecomputerbooks.com/The-SELinux-Notebook-The-Foundations.html`.

[5] Trent Jaeger. Gokyo policy analysis tool [online], May 2003. Available at `https://www.usenix.org/legacy/event/sec03/tech/full_papers/jaeger/jaeger_html/node9.html#fig:example`.

[6] Trent Jaeger, Xiaolan Zhang, and Antony Edwards. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, August 2003. Available at `http://www.cse.psu.edu/~trj1/papers/tissec2003.pdf`.

[7] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[8] Red Hat, Inc. Red hat selinux guide [online]. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/`, 2005 [cit. 2015-11-25].

[9] Beata Sarna-starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS*, pages 1–12, 2004.

[10] U. S. Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83). Available at `http://csrc.nist.gov/publications/history/dod85.pdf`.

[11] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[12] Wenjuan Xu, Mohamed Shehab, and Gail Joon Ahn. Visualization-based policy analysis for selinux: Framework and user study. *International Journal of Information Security*, 12(3):155–171, 6 2013. Available at http://sefcom.asu.edu/publications/visualization-based-policy-sacmat2008.pdf.

Appendices

List of Appendices

Appendix A

DVD Contents

The enclosed DVD contains following directory structure.

```
sepolicy_analysis/ (Selinux Policy Analysis Tool)
   +- data
      +- (Policy graph binaries)
   +- examples
      +- (Full output of examples demonstrated in Chapter 6)
   +- policy_data
      +- (SELinux policy binaries)
   +- setools
      +- (SETools 4 toolkit)
   +- bool_config       (Boolean setting configuration file)
   +- build_graph.py    (Tool for creating binary policy graph)
   +- extract_cil.sh    (Script for extracting CIL policy source files)
   +- graph_query.py    (Analysis interface)
   +- readme            (Installation instructions)
   +- security_related.conf  (Configuration file)
   +- visual_query.py   (Visual query interface)
   +- (Other non-executable python source code)
latex/
   +- (LaTeX thesis source code)
```