



Enforcing Flexible Access Control in a Networked Policy Domain

March 12, 2007

Joshua Brindle, Karen Vance, Chad Sellers

Copyright ©2007 Tresys Technology, LLC. All Rights Reserved.

Other names and brands may be claimed as the property of others. Information regarding third party products is provided solely for educational purposes. Tresys Technology, LLC is not responsible for the performance or support of third party products and does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices or products.

Abstract

Significant progress toward general acceptance of applying mandatory access control to systems has been made recently. Security Enhanced Linux (SELinux), in particular, has been enabled by default in some Linux distributions for several years. However, current SELinux deployments only effectively control a single system. Inter-system and remote resource access control capabilities are starting to appear in SELinux, but extending policy management capabilities to cover these networked systems remains an open problem. This paper discusses issues that must be addressed to support security policies distributed across a network, including policy development changes needed to be able to express a coherent security policy for a network of systems, managing the distribution of a multi-system policy, and synchronizing policy updates, and presents a distributed policy management architecture.

1. Introduction

Infrastructure to manage Security Enhanced Linux (SELinux) systems is starting to mature and user-friendly interfaces to use this infrastructure are on the horizon, but these are limited to single systems. Even in cases where multiple system management is possible, many assumptions are made about the similarity of the systems, their access matrices, and the homogeneity of the network [9] [10].

Supporting distributed systems is the next logical step for SELinux. Distributed systems are becoming much more prevalent as cheaper hardware and technologies such as hypervisors make it more economical to run many systems for redundancy and integrity than in the past. Current efforts are developing the infrastructure needed to extend security mechanisms to enforce mandatory access control (MAC) between systems and on remote objects. This will change many of the assumptions made by current SELinux deployments, but will especially affect the policy.

Current SELinux policy (i.e., the Reference Policy [1], which is now used in all SELinux distributions), assumes that all systems running the software protected by the policy are relatively the same. This may be a fair assumption for isolated systems but the ability to enforce inter-system access breaks that assumption. Therefore, the existing concept of security policy must be extended to handle a network of systems that belong within a single policy domain, but in which similar applications may be treated differently depending on the system upon which they run or the manner in which they are accessed (e.g., locally vs. from a remote machine). Within a network of systems, there will be many security servers, each responsible for providing security decisions to a set of object managers, each responsible for enforcing security decisions on their objects [12]. The objects can also be very similar, for example `/etc/shadow` on system 1 and `/etc/shadow` on system 2 are different objects controlled by different object managers.

Enforcing access control within a network of systems also introduces new challenges in policy distribution. Relevant portions of the coherent policy must be sent to the individual systems without unnecessary overhead. To this end, the policy must be partitioned and only the appropriate sections sent to individual systems. Finally, extra care must be taken in ensuring the atomicity of policy changes. While this is a concern on single systems, the problem is amplified when many systems must enforce a coherent policy on a high latency medium.

The scope of this paper is limited to a network within a single administrative domain, such as a corporate or school network. It does not address challenges related to multiple, disparate networks connecting and sharing resources.

2. Inadequacies of existing policies

Existing policies are created for a single system, i.e., they only express access control for subjects and objects on one system, whether it be an installation on a computer, in a virtual machine, or on an embedded device. The single system policy may specify network access controls, but it does not explicitly control access on subjects or objects of another system. Since the security contexts (the attributes used to identify subjects and objects to the security system) were confined to a single system policy development and analysis was always on a per-system basis. With the security contexts being extended to a network of systems, a single system policy no longer is sufficient.

2.1 Single system access control

Before delving into the challenges of multi-system access control, it is important to define existing access control in SELinux systems. Access control systems specify the allowed access between subjects, which are the active entities – i.e., processes – on the system, and objects, which are the passive entities on the system such as files. With type enforcement (TE), the primary access control mechanism in SELinux, every subject and object is given a context that comprises the security attribute used to determine what access allowed.

The Flask framework [4] upon which SELinux is based already implements distributed access control within a single system; each object manager controls its own objects and asks the security server for access control decisions. In this way, the enforcement itself is distributed throughout the system. Distributed Trusted Operating System (DTOS) [3] provides a good example of distributed access control since it is a microkernel architecture and therefore has object managers running in separate memory spaces. SELinux uses the same concepts but most of the object managers are part of the monolithic kernel. Notable exceptions are the user space object managers X Windows, DBUS, and passwd, all of which control their own objects.

Since the underlying mechanism can already support distributed access control much of the initial work to support this has already been done. The effort then falls on effectively scaling this mechanism to entire networks where before it was only effectively applied to single systems.

2.2 Expressing network-wide security goals

The first step in policy development is that of determining security goals the policy must satisfy. Existing SELinux policy assumes these goals relate to single systems and network borders. However, with security becoming more of a concern on networks, the capability to define and enforce network-wide security goals must become prevalent. A network-wide security goal may involve separating internal only and external data or limiting data access to customers and employees or even locking down workstations. One must be able to express network-wide security goals without manually determining how to enforce the goals on a per-system basis, as well as writing and distributing the policies to those systems.

Trying to develop a network-wide security policy on a per-system basis may be difficult, and the developmental effort to keep per-system policies consistent with network-wide security goals may be great. However, the most significant problem is that analysis of the policy composed of various per-system policies after development may simply be impossible. Policy analysis relies on certain aspects of the policy being true, namely that security contexts are unique equivalence classes. If a single context means something different on two different systems, analysis becomes problematic.

2.3 Analyzing policy domain interactions

Types are defined as unambiguous security equivalence classes, and are the primary security attribute used by SELinux in the TE security model. Every subject and object has a type, and any subject or object with

the same type is treated identically by SELinux. When performing policy analysis, the TE mechanism and therefore the types are the primary consideration [11].

On a single system, suppose there are two Apache web server instances, one that was for internal use only and one that was for external use. Each Apache would need different types; otherwise the internal only Apache server could be accessible externally. Separating them into different types ensures that through analysis the policy will not allow internal Apache instances to be accessible externally. If the two web server instances reside on different machines and have the same type, even though they are on different systems this guarantee cannot be made.

In some cases, systems are identically configured. In cluster systems, for example, each system would have the same policy and it is likely that each instance of an application would share the same type. This is what gives SELinux most of its flexibility. Subjects that need to be treated the same use the same type; otherwise they get a different one. Obviously, one can have some portions that should be treated identically and other portions that must be handled distinctly across systems – in the Apache example there may be an internal and external Apache that must be handled differently, but every DNS server may be the same and so the same type would be used for each of them.

In all cases, to analyze a network policy requires a single policy that covers all systems being protected. An analysis of the policy then can ensure the same guarantees that analyzing a single system policy can.

2.4 Handling distributed subject contexts

MAC has been used across systems for quite some time. Systems using the Bell LaPadula (BLP) model [6] have been able to propagate contexts to other systems for enforcement using technologies such as CIPSO and RIPS0 [2]. These technologies are of limited use for SELinux, however, as they were designed for Multi-Level Security (MLS) systems, which have inflexible policies and homogeneous security contexts. Supporting flexible MAC introduces a whole new set of issues that must be addressed.

Recent changes to SELinux have introduced the ability to propagate contexts to other systems. Using IPSEC [8] protects both the integrity and confidentiality of the communication between SELinux machines and the context that is propagated. While this is a very significant advancement, it also means additional policy development and analysis effort. The single system policy development model is illustrated to be further insufficient in the networked environment, since a context that is used to access other systems may have different security properties than the local equivalent would have.

Consider the example of a database server and client. For a client running on the same system as the server, its type might be `database_client_t`. However, if there is a client on another system that wants to access the database remotely, a decision must be made. Is the remote client equivalent to the local client? If it is equivalent, its type should be `database_client_t` on both systems. If the two were not equivalent, however, they would need different types. Suppose that the local client was primarily used for administrative purposes and therefore needed additional access; it might then instead need the type `database_admin_client_t`. Therefore, although the clients are the same application on different systems, they may need to be treated distinctly to meet the network security goals. While this may be a simple concept when considering a single client and server on two systems, extending the concept to a network of workstations with different security properties and servers with different services vastly increases the complexity of the problem. One must have a single coherent network policy in order to manage these complex relationships.

2.5 Handling distributed object contexts

In addition to subject contexts being used across systems, labeled network file systems bring the possibility of object contexts traversing the network. This means the network policy need not only treat subjects distinctively but also objects. Using the previous Apache example, if the two Apache instances are serving different data internally and externally, the data would need different types. This prevents the external Apache from accidentally or maliciously disclosing internal data whether it resides on the same system or a labeled network file system gets mounted to the system on which it runs.

Again, on single systems, assuming each Apache instance is on a different system, this would be unnecessary; on a distributed system the data and its context can be accessed on different systems, however. It would likely violate the policy domain's security goals if the external web server could access the internal data, even if the data is on a network file system that is accessible by the external server. A network-wide policy must take these matters into consideration.

3. Expression of multi-system policy

In the previous section, we determined that a single policy for the entire distributed system is necessary, rather than having individual policies for each system. Further issues arise when determining the best way to express and develop a policy in this manner.

3.1 Preserve equivalence where possible

Although types must be different between systems in some cases, the policy should not introduce unnecessary differences. Types are security equivalence classes; if two subjects or objects have the same security properties on different systems, they should use the same type. For this reason, it is not advisable to adopt a policy-wide mechanism for separating all types on a per-system basis. One should not require, for example, that the shadow file on system 1 has the type `system1_shadow_t` and the shadow file on system 2 has the type `system2_shadow_t`. If these shadow files have the same security properties, they should use a single type. Requiring per-system types would result in a policy far bigger than necessary and much more difficult to manage inter-system access on a large scale.

Instead, if two or more systems have very similar duties, e.g., load balancing mirrors or redundant routers, it is likely that they will have very similar or even identical policies. This concept can, and should, be applied on a per-application basis. As mentioned earlier, if two Apache instances need different types but the DNS servers on the same systems do not, the DNS servers should share the same type while distinct types should separate the Apache instances.

3.2 Unified namespaces

Each part of an SELinux policy – types, roles, users, etc. – has a namespace that ensures uniqueness. When a single coherent policy is used for a network of systems, these namespaces must be unique across the entire network. Any type in the policy has the same policy associated with it no matter what system is enforcing access.

This follows the type enforcement methodology of using types as equivalence classes and is conducive to policy analysis. It should also make the policies and systems generally more legible and comprehensible. A policy analysis should not have to consider whether or not a subject type, for example `httpd_t`, on one system has the same security properties as that subject type on another system.

3.3 The need for policy templating

The currently available policies do not provide a means to differentiate instances of applications other than by reproducing the entire policy for that application. Instead, this should be done via the infrastructure rather than forcing users to develop policy. One possible way to do this is to extend the policy module concept to include module templates. These would not be standard loadable modules but would include everything necessary to create an entirely new set of subject and object types with associated policies for a specific application. For example, to make a new Apache policy that is separate from the standard Apache policy, simply instantiate the httpd policy with the httpd_internal prefix. This would produce policy with httpd_internal_t as the subject type for the new Apache instance.

In addition to templating the policy, there must be a facility for adding and removing extra access or the templating is of limited use. It is unlikely that infrastructure alone can handle everything necessary to make this useful. The ability to load a module created from a template and make minor modifications would instead need to be added to the SELinux administration interfaces.

Analysis of the resulting policy would work very much the way it currently does. Since there is a single canonical version of the policy applied to the entire network, an analyst can use it to study information flow throughout the entire network instead of only a single system.

4. Multi-system policy distribution

For a single canonical version of the policy to be applied to each system in a network, an intelligent distribution scheme must be introduced. Simply sending the entire policy to each system does not scale well in large networks. A policy management server (PMS) [5] provides these intelligent partitioning and distribution mechanisms for a network. Its functionality is described in the subsequent sections.

4.1 Partitioning policy

The SELinux policy is surprisingly easy to partition into chunks. The TE policy, which is the vast majority of the SELinux policy, is keyed on the source type, the target type and the object class. With the Flask architecture, each object manager enforces access on its objects. These objects in turn are part of some object class, for example file, shm (shared memory) or even window with X Windows as an object manager. Since object managers can only enforce on their objects, they would have no need for policy relating to other object classes. In this way, policy partitions can be made according to object classes.

The object managers ask the SELinux security server for access control decisions, which in turn queries the policy. The security server, therefore, only needs policy for object classes that its object managers can enforce. If, for example, X Windows were not present on a web server, the security server on that system would not need policy for the window object class.

Since the policy uses object class as part of the key, a policy can be produced for each system that has inapplicable object classes removed, without changing the intent of the policy. If an object manager is subsequently added to a system, the policy for that system would need to be reproduced with the object classes that are necessary. Figure 4.1 shows how such a system might look.

This model also allows for multiple security servers on a single system. For example, a system may have a kernel, user space, and hypervisor security server. Each of these security servers would provide access decisions to the object managers for which they are responsible. In this case, the system's policy would be a conjunction of the object classes needed by each of the security servers for their object managers.

4.2 Distributing relevant policy

The above scheme gives the ability to remove some of the policy that is not applicable to a given system, but most systems will have the standard Linux object classes such as `file`, `dir`, `tcp_socket`, and so on, so this does not break up the policy sufficiently. A further possibility is to determine types that are applicable to a system and remove the irrelevant rules. However, it may be very difficult to determine which types are applicable to a system since new subject and object types can come from network sources.

Even without network labeling, it is not trivial to determine every type applicable to a given system. It may be possible to get a set of object types from context files and subject types from type transitions on those object types, but types set by SELinux-aware applications or administrators would not be included in the policy, which could result in a non-functioning system.

To address this issue of which types are appropriate to send to a given system, a new policy construct must be added that allows types to be bound to specific systems or groups of systems. This construct is the `@` operator, and is discussed in detail in section 5. For now however, note that this is a mechanism for determining a starting set of types needed by a given system to get it running.

Once running, the system may encounter types that were not part of its policy as a result of contexts traversing the network, either on network file systems or over labeled network sockets. If systems interact with one another frequently, they could be put in a group so that each system gets the types. In the case where interaction is not common but otherwise allowed, the security server must retrieve policy in order to know how to apply access controls to the subject or object. The appropriate policy for the unknown subject or object is fetched by the policy management server, which combines it with the existing policy for the system and sends the new policy to the system.

There is significant overhead in the policy fetch process but should be considered a corner case; the common case is that each system has all the policy it needs. Furthermore, having additional policy would use resources unnecessarily.

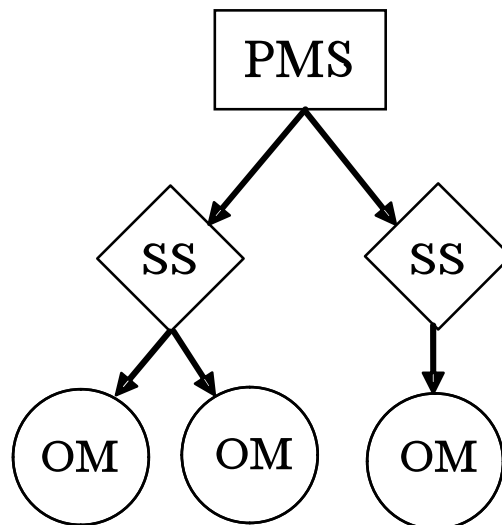


Figure 4.1

4.3 Multi-system policy management infrastructure

As Figure 4.1 showed, each object manager knows what kind of objects it manages and knows the security server it will use to get those decisions. The object managers, therefore, are responsible for informing the security server of the object classes for which it requires access decisions. The security servers then request policy from its policy management server for those object classes. The policy management server must know the SELinux identifier for its system and to which policy groups that system belongs, something that is likely to be a

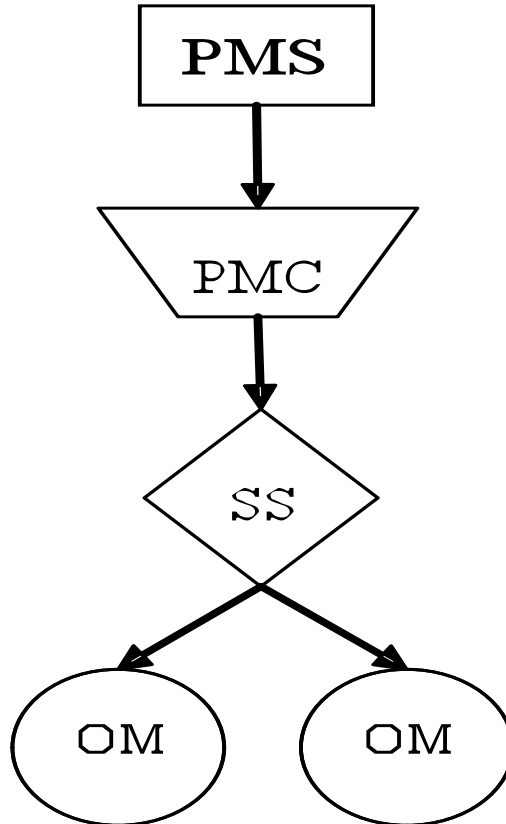


Figure 4.2

configuration setting. Unfortunately, each system would need a policy management server with this architecture.

Rather than forcing every system on the network to have policy management servers running and be policy management peers, a lighter weight daemon called the policy management client (PMC) is installed on most systems. Figure 4.2 illustrates that the PMC behaves similarly to the PMS, except that it only receives policy from a PMS and does not send policy to other systems. The PMC still manages policy for its security servers and also alerts a PMS when it needs additional policy.

The policy management by administrators or security engineers will be done with tools like `semanage` and `semodule`, which will be extended to be able to send updates to any PMS. These changes would then propagate to other PMS' and down to security servers and object managers.

5. The @ operator

As previously mentioned, the @ operator is a new policy mechanism for binding types to systems. For example, a policy author could refer to the `passwd_t` type on the `system_1` system as `passwd_t@system_1`. The operator is useful for determining which portions of policy are to be distributed to a particular security server. Additionally, it is a useful mechanism for specifying policy that is dependent on the systems on which the domains are running.

```
allow httpd_t@internal_webserver internal_www_files_t:file read;
allow httpd_t@internal_webserver external_www_files_t:file read;
allow httpd_t@external_webserver external_www_files_t:file read;

allow sshd_t@system_1 sshd_key_t@system_1:file read;
allow sshd_t@system_2 sshd_key_t@system_2:file read;
```

Figure 5.1

Figure 5.1 shows two example usages of the @ operator. The first example shows the @ operator being used to restrict access to certain internal files by Apache running on the external web server, while permitting access to those files by Apache running on the internal web server. The second example utilizes the @ operator to protect the private sshd key from access by sshd processes running on other systems. Both examples show that sometimes the location of the subject or object is an important factor in an access control decision.

The previous examples could be accomplished within the current infrastructure without creating @ a special operator. By replacing @ with a character that was permissible in SELinux type names, underscore for instance, this policy would be possible today. Type Enforcement is quite flexible in this way, so treating `sshd_t` differently on each system requires only creating types for `sshd` on each system. Although this is true, there are several advantages of creating a new operator to perform this functionality, as described in the following subsections.

5.1 Expressing multi-system policies

Although one could accomplish the policy in Figure 5.1 by simply creating additional types, the resultant policy drastically reduces the usefulness of types as equivalence classes. It would be preferable instead to have a mechanism that provided a way to write rules about types on a system, on a group of systems, or on any system. Consequently, the system name after the @ operator, such as `internal_webserver` and `system_1` in Figure 5.1, can be an individual system or a group of systems (e.g. `workstation`). These system groups are analogous to type attributes, as they contain a number of systems and can be used in rules in place of an individual system. If a rule does not specify a location, such as the target in the first three rules in Figure 5.1, that rule applies to that type on all systems.

As an additional convenience mechanism similar to the `self` keyword in the current language, the location can also be `localhost`. This provides a mechanism for expressing policy that should apply to all systems but only between types on the local system. For example, the `sshd_t` rules in Figure 5.1 would probably

need to be repeated for every system in the network. With this mechanism instead, a single rule can replace these by using `sshd_t@localhost` as the subject and `sshd_key_t@localhost` as the object.

The `@` operator provides a flexible mechanism for expressing location in a distributed policy. It extends the semantics of the policy language to incorporate the concept of location into the policy, which is critical in specifying distributed policy.

5.2 Partitioning policy with the `@` operator

In addition to providing assistance in expressing distributed policy, the `@` operator aids in distributing the policy. As stated in section 4.2, it is necessary to partition the policy into pieces that apply to a particular system. To do this, it is necessary to determine the types that apply to that particular system. The `@` operator provides a convenient mechanism for doing this – each system is given the policy applicable to the security servers it serves, including the 'global' policy, which is anything without an `@` operator, and the specific policy, which is everything with an `@` operator that specifies the system or groups of which the system is a member. This policy should be comprehensive enough to get each system up and running, and should be sufficient for all policy enforcement over local subjects and objects.

It also may be desirable for a system to request all the types applicable to a different system. If, for example, a labeled network drive is mounted, the overhead associated with requesting policy updates could be minimized if the system were able to request the policy for all objects on the remote system. This would make it unnecessary for the security server to request policy many times for individual missing pieces of the remote policy as individual files on the mounted drive are accessed.

6. Policy update synchronization

When policy changes occur, all security servers and object managers must get updated policy and apply it atomically. Failure to atomically apply policy changes could result in inconsistencies in the policy being enforced network wide.

A solution must be found to ensure that a single network policy is being enforced at all given times. This may be difficult as the network can be arbitrarily large and have a very large policy. Additionally, requests on a network can fail to arrive that would not normally fail on a single system. A logical first step is to convert all policy loading into a two-phase commit. In the first phase, each policy management client receives the policy updates. The PMC then builds the new policy and, if the build was successful, sends a confirmation of successful build. When all policy management clients have sent acknowledgment, the initiating server instructs all policy management clients to commit the changes. Until the commit occurs, each security server continues to make decisions according to known-good policy. This method would provide minimal assurance that the same policy is at least loaded into all the security servers in close time proximity, even if it isn't at the exact same time.

Further possibilities include adding serial numbers to policies so that one can ensure that each security server has the same policy loaded. This also would not ensure that the policies were loaded at the exact same time but it could be used to determine if a system was down when new policy was distributed, or for various reasons a system is enforcing an old policy.

7. Conclusions

To effectively apply mandatory access control to networks and distributed systems, the single system policy paradigm must evolve. A single coherent policy must be developed for all the systems in a policy domain to ensure network security goals are met and to allow for effective analysis. This single policy must be intelligently partitioned and distributed to send only the necessary pieces to each system, to reduce the

overhead of distributing policy over a possibly high latency network. Furthermore, all policy distribution must be synchronized so that the policy across all systems is consistent at any given time.

Finally, although the need to express distributed policies has prompted augmentation to the SELinux infrastructure and policy language, these changes will greatly facilitate the process of developing, distributing, and enforcing network policies.

8. References

- [1] "SELinux Reference Policy." Tresys Technology. <<http://oss.tresys.com/projects/refpolicy>>.
- [2] "COMMERCIAL IP SECURITY OPTION (CIPSO 2.2)." IETF CIPSO Working Group. <<http://ietfreport.isoc.org/all-ids/draft-ietf-cipso-ipsecurity-01.txt>>.
- [3] Minear, Spencer E. "Providing Policy Control Over Object Operations in a Mach Based System". Secure Computing Corporation. <<http://www.cs.utah.edu/flux/fluke/html/dtos/DOCs/usenix95.pdf>>
- [4] Spencer, Ray, Stephen Smalley, Mike Hibler, David Anderson, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. USENIX Security Symposium, Aug. 1999, University of Utah. <<http://www.cs.utah.edu/flux/papers/flaskusenixsec99.pdf>>.
- [5] MacMillan, Karl, Joshua Brindle, Frank Mayer, David Caplan, and Jason Tang. "Design and Implementation of the SELinux Policy Management Server"
<<http://selinux-symposium.org/2006/papers/01-policy-management-server.pdf>>
- [6] Bell, D. E. and L. J. La Padula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244
- [7] "Assurance in the Fluke Microkernel." Flask: Flux Advanced Security Kernel. Formal Security Policy Model. Secure Computing Corporation. <www.cs.utah.edu/flux/fluke/html/final.ps.gz>.
- [8] "Security Architecture for the Internet Protocol." Network Working Group. <<http://www.ietf.org/rfc/rfc4301.txt>>
- [9] MacMillan, Karl. Address. Tresys Technology. SELinux Symposium, Baltimore, MD. 2 Mar. 2006. na <<http://selinux-symposium.org/2006/casestudies.php#websphere>>.
- [10] Ashworth, Christopher, and James Athey. Developing SELinux Management Tools. SELinux Symposium, 14 Mar. 2007, Tresys Technology.
- [11] Mayer, Frank, Karl Macmillan, and David Caplan. SELinux by Example. Prentice Hall, 2006. 90-91.
- [12] Mayer, Frank, Karl Macmillan, and David Caplan. SELinux by Example. Prentice Hall, 2006. 39-45.