# SELinux and grsecurity: A Case Study Comparing Linux Security Kernel Enhancements

Michael Fox, John Giordano, Lori Stotler, Arun Thomas
*University of Virginia, Department of Computer Science*
*Olsson Hall*
*151 Engineer's Way*
*Charlottesville, VA 22904*
{mrf4u, jcg8f, les7j, at4a}@cs.virginia.edu

## Abstract

The security of operating systems is a main concern for all computer users and programmers. Operating system developers have addressed this concern in different ways, resulting in a number of security improvements for existing operating systems. In our paper, we report our findings from a case study of SELinux and grsecurity, two prominent Linux kernel enhancements. In implementing Mandatory Access Control (MAC), SELinux is a MAC mechanism that provides domain type enforcement and role-based access control, while grsecurity employs Access Control Lists (ACLs). We describe and quantify the differences between the two, in terms of ease of use, installation, MAC implementation, and performance. We determine that SELinux has a stronger MAC implementation, while grsecurity is simpler to use and offers other exclusive features, such as address space protection and resource limiting. We show that although SELinux and grsecurity use two different approaches to achieving the same goal, relative overall performance remains consistent.

## 1 Introduction

Herein, we report our findings from a case study conducted in order to compare the design, implementation, functionality, flexibility, usability and performance of SELinux and grsecurity. This paper not only discusses the theoretical underpinnings of these two Linux security tools, but strives to quantify their performance differentials through experimentation and testing.

In our survey of the literature, we found several comparisons between SELinux or grsecurity and one of several other tools, like Linux Intrusion Detection System (LIDS) or Rule Set Based Access Control (RSBAC), but none between the two and nothing approaching the rigor of any of the papers detailing SELinux design and implementation. We feet it would be useful to the audience to review the theory, describe different implementations and evaluate the performance of two prominent Linux security patches. We provide the reader with a comprehensive case study of these two tools in order to better inform the community of their performance potential.

### 1.1 Relevant Terminology

In this case study, we compare two security patches that include MAC mechanisms. MAC can be defined as:

> "When a system mechanism controls access to an object and an individual user cannot alter that access, the control is a *mandatory access control* (MAC), occasionally called a *rule-based access control*. [1]"

SELinux provides flexible support for policy configuration with the intention of overcoming some of the limitations of traditional MAC mechanisms.

On the other hand, MAC is implemented in grsecurity using ACLs. An ACL is a set of pairs associated with an object. Each pair contains a subject and a set of rights. Subjects can only access their associated objects using any of those rights. In grsecurity, the use of ACLs results in process-based access control. Grsecurity includes a tool, gradm, for the creation and fine tuning of ACLs. Grsecurity includes many security mechanisms besides ACLs, such as protection against buffer overflow exploits, file system protection, auditing, and randomization options.

### 1.2 Outline

In Section 2, we describe the underlying security architecture in SELinux. Section 3 provides an overview of grsecurity. In section 4, we compare the design and implementation of SELinux and grsecurity, as well as describe our experiences in installing, configuring and using each system. Section 5 discusses our results of running benchmark suites on SELinux

and grsecurity in order to quantify performance differences between them. Section 8 compares SELinux and grsecurity from a practical standpoint, explaining differences in setting up and configuring each one. The conclusions of our comparison study are outlined in section 9, and in section 10, we discuss potential future work on the subject.

## 2 Security Framework in SELinux

This section describes SELinux, including its implementation of mandatory access control.

### 2.1 Flask

Traditional MAC mechanisms have been tied to a multi-level security (MLS) policy which bases its decisions on the classification of objects and the clearances of subjects.

> "This traditional approach is too limiting to meet many security requirements. It provides poor support for data and application integrity, separation of duty, and least privilege requirements. It requires special trusted subjects that act outside of the access control model. It fails to tightly control the relationship between a subject and the code it executes. This limits the ability of the system to offer protection based on the function and trustworthiness of the code, to correctly manage permissions required for execution, and to minimize the likelihood of malicious code execution. [5]"

Because of the limitations of traditional approaches to MAC, the Flask Architecture, a general MAC architecture was created. Flask was designed to provide flexible support for security policies, giving administrators the ability to use the MAC policy model that satisfies their security requirements. It is possible to support many models of MAC, because Flask separates the security policy logic from the policy enforcement mechanism. A *security server* is constructed to hold the security policy logic and interfaces for obtaining security policy decisions. *Object manager* components are responsible for enforcing the security policy [5].

In Flask every process and object has its own security context, which stipulates all security attributes of the process or object. SELinux uses security identifiers, simply integers, to represent each security context. When a security issue arises, the enforcement code passes a pair of security identifiers (SIDs), the subject's SID and the object's SID, to a security server, which makes a decision based on the security contexts that the SIDs represent. It is important to note that the security contexts have their own user identity implementations separate from the traditional Linux user IDs (UIDs).

Flask provides an Access Vector Cache (AVC) that stores the security server's access decisions for later use. An object manager communicates with the security server to update permissions and perform permission checks. The AVC allows the object manager to communicate like this much faster, since decisions are cached.

Flask encapsulates security labels, supports flexibility in labeling and access decisions, and supports policy changes. It has a process management component and provides a mechanism for controlling access to whole file systems, individual files, and directories [9].

### 2.2 SELinux as an Application of Flask

SELinux is simply an application of the Flask architecture in Linux. The policy control mechanisms implemented in SELinux are referred to as "permission checks that have been inserted at control points throughout the Linux kernel." About 140 permissions, grouped into 28 object classes, are defined and make almost every system operation controllable. The permission checks communicate with the security server to access SIDs and determine access amongst instances of objects in the system.

In [6] a description of an example security server implemented for SELinux is given. This example uses a combination of Identity-based Access Control (IBAC), Role-based Access Control (RBAC), and Type Enforcement (TE). A security context in SELinux has three attributes: an identity, a role, and a type. Every process in the system has an identity. This identity is separate from standard Unix user id. The SELinux user identity determines what roles and domains can be used. A user of the system has a set of roles that he may take on at any time. For example, an administrator will use the user role (user_r) when he is doing normal user tasks, but he will switch to a more privileged admin role (sysadm_r) when he must add users and the like. It is important to note that when the user switches roles, his identity does not change.

The role determines what domains can be used. The policy determines which roles each identity can become. Every object in the system has a type, which determines who can access the object. Every process runs in a domain. The domain determines how much access a process has. In SELinux *types* are synonymous to *domains*. The concept of a domain and a type are similar except types apply to object (e.g. files, sockets), whereas, domains apply to processes. The RBAC policy in SELinux allows a user to enter some specific domain by way of an individual role in his set of roles. The TE policy allows for fine-grained access control.

SELinux provides several security mechanisms including ones for process control, file control, and socket control. Sockets are accessed through file descriptions and therefore inherit permissions defined for the file object classes. Refer to the section entitled Security Mechanisms in [5] for a more detailed description of the security mechanisms implemented in SELinux.

## 2.3 Policy Configuration

### 2.3.1 The Security Policy Language

With SELinux one can implement a security model within the context of a combination of a TE model and an RBAC model, as described earlier. With a TE model one can define the security policy for processes and objects at a very low-level, and with the RBAC model one can maintain a higher-level abstraction of the policy for management ease.

TE statements are attribute declarations, type declarations, type transition rules, type change rules, access vector rules, or assertions. All of these describe attributes and rules of types that one can create. RBAC statements are role declarations, role dominance definitions, or role-allow rules. They describe rules for user roles that one can create. The TE and RBAC grammar is as follows:

```
te_rbac -> te_rbac_statement | te_rbac
te_rbac_statement
te_rbac_statement -> te_statement |
rbac_statement
te_statement -> attrib_decl |
                type_decl |
                type_transition_rule |
                type_change_rule |
                te_av_rule |
                te_assertion
rbac_statement -> role_decl |
                  role_dominance |
                  role_allow_rule
```

[7]

More specialized grammars are available making the SELinux policy language robust and complete. These include grammars for attribute declarations, type declarations, role declarations, role transition roles, etc. Below are some applications of the grammar that show how the TE and RBAC models interact.

```
user root roles { staff_r sysadm_r }
```

This example shows how roles map to an identity. According to this rule, the root user can operate in the *staff_r* and *sysadm_r* roles.

```
allow system_r sysadm_r;
```

This rule allows the system_r role to transition to the sysadm_r role.

```
type lib_t file_type, sysadmfile;
```

This rule defines a type lib_t for system library files. Only the administrator should be able to alter these files, so this type is designated with sysadmfile modifier.

```
/sbin/insmod.*
system_u:object_r:insmod_exec_t;

allow sysadm_t insmod_exec_t:file
x_file_perms;
allow sysadm_t insmod_t:process transition;
allow insmod_t insmod_exec_t:process
{entrypoint execute};
allow insmod_t sysadm_t:fd inherit_fd_perms;
allow insmod_t self:capability sys_module;
allow insmod_t syadm_t:process sigchld;
```

These examples [5] show a portion of the *insmod* utility policy. The *insmod* utility allows a system administrator to insert kernel modules. The *insmod* program is labeled with the *insmod_exec_t* type as shown in the first example, though it runs in the *insmod_t* domain. The first example defines the context for the *insmod* utility; it has the *system_u* identity, *system_r* role, and it resides in the *insmod_exec_t* domain. In the second example, the first rule allows the administrator to execute the *insmod* program. The second rule allows *sysadm_t* domain to transition to the *insmod_t* domain. The third rule allows the *insmod* program to enter the *insmod_t* domain (by declaring the entry point) and execute in that domain. The fourth rule allows the *insmod* utility to use file descriptors from *sysadm_t* domain. The fifth rule allows the insmod utility to use the *CAP_SYS_MODULE* utility, which allows for inserting and removing kernel modules. The final rule allows *insmod* to send the *SIGCHLD* signal to the *sysadm_t* domain when it exits.

### 2.3.2 Policy Customization

There are degrees of difficulty in customizing the policy. For example it is fairly easy to add a user to the system; the administrator must edit the *policy/users* file by specifying a name for the user and the user's associated allowable roles. Adding permissions, however, is more difficult because first new rules must be defined and it may be necessary to make modifications to the RBAC model for the object or subject in question or even relax or tighten constraints with the involved object or subject.

## 2.4 Summary

SELinux is an application of the Flask architecture. It provides a MAC mechanism that incorporates IBAC,

RBAC and TE. The policy syntax is non-trivial, but allows for flexible policy configuration.

# 3 Grsecurity Overview

Grsecurity is a suite of patches (300K total) that is an attempt to improve Linux security. According to Brad Spengler, the creator of grsecurity, the suite meets four goals. First, grsecurity offers configuration-free operation. Second, it gives protection against all kinds of address space modification bugs. Next, grsecurity includes a rich access control list system and many auditing systems. Finally, it operates on multiple processor architectures and operating systems.

As stated by Spengler, there are many problems with the current "avoid/identify/fix" method of dealing with software bugs. He likens the task of keeping systems secure to a "never ending rat race," an endless cycle of discovering and fixing bugs. Grsecurity is offered as a solution, and is reported to detect, prevent, and contain software bugs that are security vulnerabilities. Detection is obtained through auditing and logging of attacks. Prevention is implemented by PaX (address space protection) and other techniques. Finally, containment is offered by grsecurity's access control list system. The following sections discuss this ACL system and the various other security options offered by grsecurity.

## 3.1 Grsecurity Access Control Lists

Mandatory access control is implemented in grsecurity using access control lists (ACLs) [10]. Within these structures, administrators define restrictions on subjects, including access to files, capabilities, resources, and IP. For every event, the kernel will check the ACL for the executing process and the standard Linux ACL with the requested object. Access is granted only when both systems agree.

Grsecurity ACLs are made up of subjects (processes) and objects (files, capabilities, resources, and IP ACLs). ACL structures define what the restrictions that processes should adhere to. Inheritance is provided to reduce the necessary configuration needed for similar binaries.

ACLs have the following general structure:

```
<path of subject process> <optional subject
modes> {
        <file object> <optional object modes>
        [+|-]<capability>
        <resource name> <soft limit> <hard
        limit>
        connect {
                <ip>/<netmask>:<low port>-<high
                port> <type> <proto>
                 }
        bind    {
                <ip>/<netmask>:<low  port>-<high
                port> <type> <proto>
                }
}
```

This implementation of ACLs creates a form of process-based mandatory access control. It is possible to restrict what a process can and cannot do. Additionally, access can be restricted to an object for any user, even root. Further, these restrictions cannot be changed by normal users. The system will soon offer role-based access control as well [10].

### 3.1.1 IP Access Control Lists

Grsecurity IP access control lists allow administrators to control many things, such as what IPs and ports a process can bind to on a server, what IPs and ports users can connect to remotely, what kind of sockets a process can use, what protocols sockets are allowed to use.

As shown above, the format of an IP ACL is:

```
connect{
        <ip>/<netmask>:<low port>-
        <high port> <type> <proto>
        }
bind    {
        <ip>/<netmask>:<low port>-
        <high port> <type> <proto>
        }
```

For example, a valid IP ACL is [10]:

```
connect{
        192.168.1.2/24     stream dgram tcp udp
        134.55.22.12/24:80   stream tcp
        }
bind    {
        192.168.1.2/24:1024-
        65535 any_type any_proto
        }
```

### 3.1.2 Gradm tool

With grsecurity comes a powerful tool called gradm. This tool is used for configuring ACLs. Specifically, it parses ACLs, enforces a secure base policy, optimizes ACLs, and offers a learning mode for fine-tuning ACLs.

Learning mode in grsecurity is process-based. It can be used on a single process while the rest of the system remains protected. It can be used to create an access control list that is optimized for a new process on a particular environment. Learning mode supports files, capabilities, resources, and socket usage.

## 3.2 Additional Security Mechanisms

Grsecurity includes many security mechanisms besides access control lists, such as protection against buffer

overflow exploits and protection against fork bombs. It includes PaX, logging options, executable protections, network protections, and more. This section will explore those mechanisms. In the following subsections, the italicized text represents grsecurity configuration options.

### 3.2.1 Filesystem Protection

In Linux, the /proc filesystem is a pseudo-filesystem used as an interface to kernel data structures. Although most of it is read-only, some files allow kernel variables to be changed [2]. Sometimes, it is subject to exploits, so grsecurity provides *Proc restrictions*.

While using these filesystem protections, multiple restrictions are available. First, *restrict to user only* ensures that users can only view information about their own processes. *Additional restrictions* are available to hide CPU and device information. For users of chroot jails, *chroot jail restrictions* offers several options for chroot restrictions. Additionally, *Linking restrictions* control a user's ability to follow symbolic links and make it impossible to hardlink to files she does now own. Finally, *FIFO restrictions* curb users from writing to FIFOs in world-writeable directories if they are not the FIFO or directory owner [4].

### 3.2.2 Kernel Auditing

Grsecurity allows administrators to configure the amount of logging provided by the kernel [4]. This auditing capability is meant to detect attacks. Examples of audited events include exec, chdir(2), mounting/unmounting devices, signals, failed forks.

### 3.2.3 Executable Protections

Executable protection is provided in grsecurity, since most exploits work through or with running processes. If *Enforce RLIMIT_NPROC* is enabled, resource limits on processes are checked during execve() calls. *Emesg(8) restrictions* prevent non-users from using *dmesg* to view the log buffer. Further options include *Randomized PIDs* and *Trusted path execution*.

### 3.2.4 Network Protections

To prevent the potential prediction-based attacks in the default Linux TCP/IP stack implementation, grsecurity includes many options. *Larger entropy pools* doubles the poolsize. *Randomized IP IDs* are available to prevent operating system fingerprinting and spoofed scans. To prevent RPC connection hijacking, *Randomized RPC XIDs* can be enabled to randomize RPC transaction IDs (XIDs). With *Truly random TCP ISN selection*, TCP Initial Sequence Numbers are randomized. *Randomized TCP source ports* randomizes the dynamically generated connect() source port. With *Altered Ping IDs*, ICMP echo replies are altered to make their IDs equal to the ID of the echo requests they respond to. Socket restrictions are offered to *Deny any sockets to group*, *Deny client sockets to group*, and *Deny server sockets to group* [4].

### 3.2.5 PaX Address Space Protection

Many Linux exploits, such as buffer overflow bugs, take advantage of how Linux handles memory. The PaX project attempts to prevent and contain the problem by creating defense mechanisms against exploits that give an attacker access to the attacked task's address space [3]. This section will contain a brief overview of PaX.

New executable code can be introduced into a task's address space in two ways. First, an executable mapping can be created. Second, an existing writable/executable mapping can be modified. PaX attempts to deal with the second method, while leaving the first to access control mechanisms.

To combat the problem of writable/executable mapping modifications, PaX provides the NOEXEC category of features. The philosophy of NOEXEC is that if some data in a task's address space does not need to be executable, then it should not be. These pages must be marked as non-executable. Furthermore, if an application does not need to generate code at runtime, then it should not have the ability to. PaX should, therefore, be able to prevent memory page state transitions between executability and writability. So, NOEXEC enforces a type of least privilege. Some of the features of NOEXEC are that it implements executable semantics on memory pages, makes the stack and heap non-executable, creates ELF (Executable and Linking Format) object file mappings with only the requested access rights (only those with code will be executable), and locking down of permissions on memory pages.

The Linux implementation of PaX is split into two main feature sets, NOEXEC (PAGEEXEC and SEGMEXEC), and the MPROTECT page protection restrictions. PAGEEXEC implements the non-executable page feature using the paging logic of IA-32 based CPUs. SEGMEXEC, on the other hand, implements the non-executable page feature using the segmentation logic of IA-32 based CPUs. MPROTECT attempts to prevent the introduction of new executable code into the task's address space.

In practice, most attacks require advance knowledge of assorted addresses in the attacked task. Consequently, PaX contains a second set of features, the address layout randomization (ASLR) features. ASLR is intended to introduce randomness into these addresses,

thus forcing attackers to guess each address or obtain it by brute force. With this option activated, exploits will probably crash the attacked application, making it easy to catch and react to the attack [4]. Options are available for randomizing the kernel stack base, randomizing the user stack base, and randomizing the nmap() base.

It should be noted that some applications need to do things that PaX disallows. The provided tool *chpax* gives the user fine grained control over PaX features on a per executable basis to combat this problem.

PaX is part of grsecurity. By default, PAGEEXEC, SEGMEXEC, MPROTECT, and RANDMAP are all enabled on ELF binaries in the grsecurity system [10]. Additionally, PaX has been decoupled from grsecurity and can be integrated into SELinux and other MAC projects.

### 3.3 Summary

Grsecurity is a suite of patches that attempt to improve Linux security in many ways. First, it offers an ACL system. Second, it employs auditing and logging for detection of attacks. PaX is used to prevent attacks by protecting the address space. The combination of these security mechanisms detect, prevent, and contain attacks.

## 4 Security and Application Comparison

In this section, we compare the SELinux and grsecurity implementations of mandatory access control mechanisms. We also describe and compare our experiences with installing and using the two.

### 4.1 Protection Models

Grsecurity has a simple MAC implementation. It does not implement the domain-type enforcement component of SELinux, though the grsecurity MAC functionality is somewhat similar. Both essentially provide an access matrix defining permissions between processes and files. Domain-type enforcement is more flexible in terms of policy definition. It provides for better isolation between processes and allows for a more fine-grained description of the sharing between processes.

Grsecurity has no concept of role-based access control, but we can expect to see RBAC in a subsequent release. Consequently, grsecurity does not allow the administrator to give different levels of access to different non-root users outside the limited discretionary access control (DAC) mechanisms (in which an individual user can allow or deny access to an object [1]). Both grsecurity and SELinux will respect DAC if DAC permits less access than the respective MAC implementation.

SELinux also provides more fine-grained access than grsecurity in general, especially with respect to sockets and other inter-process communication (IPC) mechanisms. On the other hand, grsecurity supports certain features that SELinux lacks. For example, grsecurity supports fork() rate limiting, various resource limiting and randomizing options.

Grsecurity also includes PaX for protection against stack smashing attacks and the like. PaX has recently been updated so it may be used on SELinux systems, so this is no longer an advantage.

The path-based protection mechanism that grsecurity provides is considered weak, since paths are not necessarily unique. SELinux uses an inode-based approach, so it does not have this weakness.

### 4.2 Installation

Grsecurity is much simpler to install than SELinux. To install grsecurity, the site administrator need only patch the kernel with the grsecurity kernel patches and install gradm. If she wants to employ the PaX address space protection mechanism, she must also install the *chpax* utility. No other changes need be made.

To install SE Linux, the site administrator must patch the kernel. She must also install libselinux, checkpolicy, and policycoreutils. These packages are roughly equivalent to gradm in that they provide the userspace component of SELinux. Additionally, various system related utilities such as *login* and *ps* must be patched in order to support security labels.

If the administrator uses a Linux distribution that supports SELinux such as Gentoo or Debian however, the SELinux installation process is comparable to grsecurity in difficulty. The distributions take care of installing the SELinux utilities and patching the necessary applications.

### 4.3 Ease of Use

Overall, grsecurity is simpler to administer than SELinux. First, grsecurity policies are simpler to create, since there are no roles or complicated domain/file transitions. Second, the administrator need not write policies manually. Gradm in learning mode can be used to generate policies automatically. The administrator will likely want to adjust these policies somewhat.

There is no such tool for SELinux. SELinux ships with a very simple perl script audit2allow that will convert denials in system logs to SELinux rules. This tool requires that a basic policy exists. It can only be used to adjust an existing policy. Since the script performs such

a simple translation, the user does not gain much from using it aside from a few saved keystrokes. The generated rules must be audited carefully, as the script does not analyze the security impacts of the generated rules. Tresys has released some tools to help with policy analysis and construction yet, these tools do not generate policies automatically either.

SELinux requires that all files be labeled with a security context. Whenever an administrator installs a new program, she must manually re-label all the files associated with the context. Gentoo and Debian both provide package managers that support for automatic file re-labeling. SELinux requires that the administrator use SELinux wrappers for standard UNIX user management utilities, such as *useradd* and *vipw*. Grsecurity relies on the standard utilities. Unlike grsecurity, SELinux also requires that the administrator creates an initial ramdisk to hold the policies. This is not an overly onerous requirement, however.

## 4.4 Documentation

Probably because it was developed by the NSA, an abundance of documentation is available for SELinux. This documentation includes many published papers, technical reports, presentations, and mailing lists.

Grsecurity, unfortunately, is not as well documented. Only one formal document is available, and it focuses on only the ACLs [10]. Additional information can be found in forums and in informal web sites.

## 4.5 Linux Security Modules

The Linux Security Modules (LSM) project is an attempt to include a security framework within the mainstream Linux kernel. The project came about after many projects, such as SELinux, grsecurity, LIDS, DTE, and SubDomain, developed security kernel patches for Linux [8].

Basically, the LSM kernel patch provides a framework to support access control modules. Alone, the framework does not provide extra security. The patch adds security fields to kernel data structures. Then, calls to hook functions are inserted at critical points in the kernel code to manage these security fields and enforce access control. Functions are added for registering and un-registering security modules. This infrastructure can then be used by loadable kernel modules to implement any desired model of security. At the present, LSM only focuses on access control, but it may be extended in the future [8].

SELinux and grsecurity have taken widely different stances on the LSM project. Although SELinux was originally developed as a kernel patch, it was totally reimplemented as a security module using LSM.

Grsecurity, on the other hand, does not use LSM. This is for multiple reasons. First, Spengler believes that LSM could be detrimental to Linux security. He says, "Because LSM is compiled and enabled in the kernel, its symbols are exported. Thus, every rootkit and backdoor writer will have every hook he ever wanted in the kernel. This will allow for a new generation of sophisticated backdoors and rootkits that will be nearly impossible to detect [11]." Additionally, Spengler says that LSM is not appropriate for grsecurity because it only involves access control; the additional features of grsecurity would not operate under LSM.

The use of LSM remains a fundamental difference in the design of the two patches.

# 5 System Performance Expectations and Benchmarking

As we have discussed, SELinux and grsecurity differ in their respective security models. Since security mechanisms generally have some performance impact, we wish to quantify how these two implementations differ in terms of performance. As in [12], the best way to evaluate performance is through objective benchmarks. We chose to replicate the benchmarking tests conducted in [5] and included grsecurity data.

We note that we have replicated the tests and not the experimental conditions under which Loscocco and Smalley conducted their tests of SELinux. Our intent is to neither validate nor refute their findings, but rather to provide the reader with a comparative basis by which one can asses these two implementations critically as a component of our case study. While performance differentials exceed the bounds expressed in [5], in some cases by orders of magnitude, our results reflect performance based on our particular experiment's environment.

Microbenchmarking suites such as lmbench and Unixbench perform a sequence of low-level operations in a controlled, timed environment and report performance metrics. As in [5], we converted the units in which Unixbench reports its results to units that compare more readily to lmbench results. The two benchmarking suites both employ a number of sub-programs, or microbenchmarks, that perform low-level operations. The low level of these tests provides precision required to make informed conclusions regarding performance.

We established a testbed consisting of two identical PCs, each configured with Gentoo Linux 1.4.1, running AMD K6 II microprocessors with a clock speed of 450MHz, and having 8Gb Fujitsu hard drives and 128Mb of memory. One PC's Linux installation was

patched with SELinux, and the other grsecurity. We conducted three iterations of each test suite on both SELinux and grsecurity with security policies being enforced in both.

Before conducting benchmark experiments, we surveyed the literature and posed a hypothesis regarding performance, based upon the theoretical and practical concerns presented by both models and implementations. Since system performance is derived from a relatively high-level view, our hypothesis contended that there would be no significant overall difference in observed system performance between SELinux and grsecurity, but that there would be differences in their performance of specific tasks, like disk operations, I/O, floating point calculations etc., largely as a result of how each patch implements its security model at low levels. Since benchmark results are most often used as a basis for decision-making, we leave it up to the reader to apply the results and interpretations that we have included here.

## 5.1 Unixbench

Our results for Unixbench are shown in Table 1. The tests we chose to include correlate to most of the tests in [5], but we note they are only a subset of all results available for analysis. We feel the variety of microbenchmark tests in terms of granularity of operations provides for an acceptable basis of comparison and should underscore the performance differentials between the two patches.

| Microbenchmark | Base | SELinux | Over-head | grsec | Over-head |
|---|---|---|---|---|---|
| File Copy 4096 | 30.5 | 29.4 | **3.4%** | 20.4 | **32.9%** |
| File Copy 1024 | 28.7 | 28.7 | **-0.1%** | 30.5 | **-6.4%** |
| File Copy 256 | 14.3 | 13.6 | **5.1%** | 16.0 | **11.7%** |
| Pipe | 3.3 | 5.0 | **52.9%** | 3.2 | **-3.6%** |
| Context Switching | 10.4 | 16.3 | **56.7%** | 10.2 | **-1.7%** |
| Process Creation | 451.4 | 477.3 | **5.7%** | 477.5 | **5.8%** |
| Execl | 1379.9 | 1371.6 | **-0.6%** | 1390.0 | **0.7%** |
| Shell Scripts | 365.5 | 383.2 | **4.9%** | 348.4 | **-4.7%** |

**Table 1 – Unixbench Results**

The file copy operations capture the number of characters copied to a file based on the various buffer sizes indicated. Pipe measures inter-process communication. Context switching captures the communications between a parent and child process. Process creation measures the number of child process that can be forked in 10 seconds. Execl replaces a currently running process with a new one. The shell script test measures the execution of a shell script by 8 concurrently running processes.

We observe conflicting results for file copy operations. In SELinux, as buffer size increases, latency becomes negligible, whereas in grsecurity, an increasing buffer size induces a penalty of increasing magnitude. We observe significant performance penalties in SELinux for pipe throughput and pipe-based context-switching as a result of the revalidation of permissions

Since subject-object associations must still be verified, this results in additional overhead relative to grsecurity where the ACL is loaded and checked once. In the remaining microbenchmarks, we see negligible overhead or even improved performance relative to the baseline consistently with both patches.

## 5.2 lmbench

Our results for Lmbench are shown in Table 2. The tests included here, as in Unixbench, are a subset of all microbenchmark tests available for analysis from the suite.

| Microbenchmark | Base | SELinux | Over-head | grsec | Over-Head |
|---|---|---|---|---|---|
| null I/O | 1.78 | 3.72 | **108%** | 1.52 | **14%** |
| stat | 7.74 | 15.88 | **105%** | 293 | **3.6K%** |
| open/close | 9.90 | 19.60 | **98%** | 1098 | **11K%** |
| fork | 486 | 464 | **-4.69%** | 484 | **0.58%** |
| evecve | 1533 | 1543 | **0.64%** | 1611 | **5.1%** |
| sh | 6321 | 6482 | **2.55%** | 6379 | **0.92%** |
| pipe | 9.83 | 15.88 | **61.53%** | 9.24 | **6.1%** |
| AF_UNIX | 19.07 | 24.67 | **29.38%** | 21.42 | **12.3%** |
| UDP | 48.1 | 45.5 | **-5.38%** | 42.2 | **12.1%** |
| TCP | 62.1 | 83.2 | **34.%** | 71.0 | **14%** |
| TCP/IP | 248 | 292 | **17.5%** | 267 | **7.4%** |

**Table 2 - lmbench results**

As in [5], null I/O is the average combined time for a one byte read and write operation. The stat test measure the time required to obtain the status of an unopened temporary file created by the benchmark suite. Open/close simply times the opening and closing of a temporary file for reading. Fork, execve and sh time process creation from fork() and exit, to fork() and execve, to fork() and instantiating the shell. Pipe latency, AF_UNIX, UDP and TCP latency and TCP/IP tests all time inter-process communication between two processes on the tested system.

We observe pronounced overhead in SELinux in the for the I/O timing, largely as a result of having to revalidate permissions for each operation in SELinux. In grsecurity, however, we see a performance degradation orders of magnitude more than in SELinux in file operations, but not in IPC, and certainly not with the same magnitude. This latency is likely a result of repeated checks of the ACL in grsecurity for each file operation.

## 5.3 Conclusions

In general, we have not seen a significant gap in overall system performance between the two patches, but our benchmark suites do reveal some critical performance advantages along with some inconclusive results. Unixbech's microbenchmarks that target inter-process communication reveal noticeable overhead in SELinux relative to grsecurity, yet lmbench's tests targeting the same do not reveal that, at least not conclusively. This disparity makes it difficult to asses each patch's relative performance in this domain. Perhaps underscoring more of the subjectivity of interpreting microbenchmark results at a high level, our experiments cannot provide the basis for making an implementation decision with regards to either SELinux or grsecurity, nor can we predict performance based on our results. Our hypothesis before experimentation is largely validated, and we have revealed the metrics that support our claim that different processes, and hence user tasks, will incur overhead based on the security model being implemented. Our metrics and conclusions presented in this section say nothing about the inherent security afforded by either patch.

# 6 Policy Analysis and Experimentation

In order to more effectively compare grsecurity and SELinux, we decided to write a security policy for a representative application. We chose irssi, a popular command-line IRC client, since it is not overly complex but somewhat representative of networked applications that run on server machines.

## 6.1 Grsecurity policy

We generated the grsecurity ACL by running gradm in learning mode. Gradm would then analyze the system logs to see what resources the application required and output a policy.

In order to use gradm, we first created a default least privilege ACL in learning mode. Below is the policy which was generated after running irssi several times.

irssi.acl:

```
/usr/bin/irssi o {
        /usr/share/zoneinfo/US/Eastern r
        /usr/share/terminfo/l/linux r
        /usr/share/irssi/themes/default.theme r
                /usr/lib/perl5/vendor_perl/5.8.
        1/i586-linux/Irssi/Irc.pm
        /usr/lib/perl5/site_perl/5.8.0
        /usr/lib/perl5/5.8.1/Symbol.pm r
        /usr/lib/perl5/5.8.1/Exporter.pm r
        /usr/lib rx
        /proc/sys/kernel/version r
        /proc/1941/exe
        /proc/1941
        /proc/1885/exe
        /proc/1885
        /proc/1880/exe
        /proc/1880
        /lib rx
        /lib/ld-2.3.2.so x
        /home/lori/.irssi/config r
        /home/lori/.irssi
        /etc r
        /dev/urandom r
        /dev/null r
        /usr/bin/irssi x
        / h
        -CAP_ALL
        RES_FSIZE 0 0
        RES_DATA 587536 587536
        RES_STACK 17384 17384
        RES_RSS 0 0
        RES_NPROC 8 8
        RES_NOFILE 8 8
        RES_MEMLOCK 54096 54096
        RES_AS 7082272 7082272
        RES_LOCKS 0 0

        connect {
                130.239.18.172:6667 stream tcp
                209.218.71.2:6667 stream tcp
                128.143.136.15:53 dgram udp
        }

        bind {
                0.0.0.0:0 dgram ip
        }

}
```

The policy defines which files the program can access and what sort of access is allowed. It also determines what sort of resources the process is allowed. Finally, it determines the application's allowed network operations.

We modified the above policy by hand in order to allow any user to use irssi. In the gradm-generated policy, only the user lori would be able to run irssi. The irssi binary was allowed access only to the user lori's configuration files. Additionally, the gradm-generated policy was modified to allow the irssi executable to connect to any IRC server. Finally, the ACL was modified to allow access to any /proc filesystem entry that might be used. These manual changes were necessary, since we did not run the program in all possible scenarios during the learning phase.

## 6.2 Selinux policy

We developed this policy mostly by hand through careful examination of the system logs for AVC denials while irssi was running. We consulted other example policies for guidance. We also used the audit2allow script for minor tweaking. We carefully audited the rules audit2allow generated, however.

irssi.fc:

```
# irssi
/usr/bin/irssi
         system_u:object_r:irssi_exec_t
```

This file sets up the security context of the irssi binary. The file contexts for irssi's configuration files are defined by the default policy. The file /etc/irssi.conf has the context system_u:object_r:etc_t; whereas, the file <userid>/.irssi/config has the context <userid>:object_r:user_home_t.

```
irssi.te:

#DESC irssi - IRC client
#

user_application_domain(irssi)
can_network(irssi_t)

# lib access
allow irssi_t lib_t:file { getattr read ioctl
};

# allowed signals
allow irssi_t irssi_t:process { signal fork
sigchld };

# use of proc filesystem
allow irssi_t proc_t:dir { search };
allow irssi_t proc_t:lnk_file { read };

# access config files
type user_home_irssi_t, file_type, sysadmfile;
allow irssi_t home_root_t:dir { search getattr
};
file_type_auto_trans(irssi_t, user_home_dir_t,
user_home_irssi_t, dir)
file_type_auto_trans(irssi_t, user_home_t,
user_home_irssi_t, file)
allow irssi_t user_home_t:file { getattr read
write };
allow irssi_t etc_t:file { getattr read };

# locale support
read_locale(irssi_t)

# name resolution
allow irssi_t resolv_conf_t:file { read
getattr };

# access urandom
allow irssi_t random_device_t:chr_file { read
};

# pts support
allow irssi_t user_devpts_t:chr_file
rw_file_perms;

# ssh
allow irssi_t sshd_t:fd { use };

# access theme
allow irssi_t usr_t:file { getattr read };

# needed for pipe
allow irssi_t irssi_t:dir { search };
allow irssi_t irssi_t:fifo_file { read write
};

# other denials
#allow irssi_t bin_t:dir { search };
#allow irssi_t opt_t:dir { search };
#allow irssi_t sysctl_kernel_t:dir { search };
#allow irssi_t sysctl_kernel_t:file { read };
#allow irssi_t sysctl_t:dir { search };
#allow irssi_t irssi_t:lnk_file { read };
```

The first two lines are macros. The first sets up various operations associated with the a user domain application. For example, it creates the irssi_t and irssi_exec_t domains and defines the transitions to and from these domains. The second macro defines common networking operations. There are a few other macros that ease the policy creation policies. The file_type_auto_trans macro is of special interest. If the creating domain of the file is irssi_t and the parent directory of the file is user_home_t (or user_home_dir_t), the file will be given the user_irssi context. The rest of the lines determines what operations (e.g. Read) are valid on each type. The lines at the end of the file are commented, since they are not actually necessary to run irssi. The program does attempt these operations, however, so they will be logged as AVC denials in the system logs when these lines are commented.

# 7 Conclusions

After doing a thorough theoretical and practical comparison between SELinux and grsecurity, we were able to make several broad conclusions about the potential advantages and disadvantages of each system with respect to the other. We compared the two in terms of their theory and practicality so as to provide a deeper understanding of their differences.

SELinux is a more powerful access control mechanism, since it incorporates role-based access control and more fine-grained access control in general. Nevertheless, we believe the two theories underscore sound security models. They both allow for easy control of access between processes and objects, processes and other processes, and objects and other objects.

While both systems implement MAC, they have many distinguishing characterisitics. The Flask architecture in SELinux provides for a flexible security policy. The policy language is complex but allows for powerful security configurations. Grsecurity, on the other hand, comes with the *gradm* tool, which is capable of programmatically optimizing and fine-tuning ACLs in the operating system. When making a decision about implementing one of these systems, one should consider that SELinux provides a more powerful access control mechanism, whereas grsecurity is easier to use and includes many other security options. In terms of relative performance, while we observed differences in microbenchmark measurements, both systems' overall performance was similar.

## 8 Further Work

In our experiments, we ran a number of microbenchmarks to compare the performance of grsecurity and SELinux. We would like to run more of these microbenchmark suites to gather further performance data. Additionally, we would like to run various real-world applications such as Apache or Bind.

Moreover, we would like to further explore the many security options of grsecurity. For this work, we largely focused on access control and PaX because many of grsecurity's security mechanisms are not available in SELinux.

Furthermore, we would like to compare the next generation of grsecurity against SELinux. It may prove to be a more interesting comparison, since it is slated to support RBAC. It would also be interesting to compare RSBAC, a less popular MAC system that incorporates both RBAC and DTE. We would also like to investigate the SELinux policy tools from Tresys. Additionally, we would like to do a vulnerability assessment of systems running both SELinux and grsecurity. Finally, we would like to measure the performance of real-world program, such as apache.

## References


[1] M. Bishop. *Computer Security Art and Science.* Addison-Wesley, 2002.

[2] D. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly & Associates, 2002.

[3] Documentation for the PaX Project. `http://pax.grsecurity.net/docs/index.html`.

[4] Gentoo Linux grsecurity Guide. `http://www.gentoo.org/proj/en/hardened/grsecurity.xml`.

[5] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference,* pages 29-42, Boston Massachusetts, June 2001.

[6] P. Loscocco and S. Smalley, "Meeting Critical Security Objectives with Security-Enhanced Linux", In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[7] S. Smalley. Configuring the SELinux Policy. Technical Report 02-007, NSA and NAI Labs, February 2002.

[8] S. Smalley, T. Fraser, and C. Vance. Linux Security Modules: General Security Hooks for Linux. `http://lsm.immunix.org/docs docs/overview/linuxsecuritymodule.html`.

[9] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123-139, Aug. 1999.

[10] Brad Spengler. Grsecurity ACL Documentation V1.5, April 1, 2003.

[11] Brad Spengler. LSM. `http://www.grsecurity.net/lsm.php`.

[12] Rodney C. Wilson. Unix Test Tools and Benchmarks. Prentice-Hall. Upper Saddle River, NJ 1995. 152-155.