

# SELinux and grsecurity: A Side-by-Side Comparison of Mandatory Access Control and Access Control List Implementations

Michael Fox, John Giordano, Lori Stotler, Arun Thomas  
{mrf4u, jcg8f, les7j, at4a}@cs.virginia.edu

## Abstract

Improvement of the security functionality of operating systems is a main concern for all users and programmers of computers. Operating system developers have addressed this concern several ways. In our paper we discuss two approaches, SELinux and grsecurity, to improving the security of the Linux operating system. Being highly regarded implementations of an enhanced-security Linux kernel, these two approaches lend themselves to a comparison study. Through our comparison study we strive to educate our audience on the state-of-the-art of operating system security enhancement and document results and conclusions that express the advantages of one approach over the other.

We describe SELinux's implementation of the Mandatory Access Control (MAC) Model and grsecurity's implementation of access control lists (ACLs) and the MAC Model.

The National Security Agency (NSA) together with the Secure Computing Corporation (SCC) assessed the limitations of traditional MAC and developed the Flask architecture designed to overcome those limitations. They implemented Flask in Linux to produce SELinux. With Flask, users can customize security policies very easily and enforce them with several provided security mechanisms.

Grsecurity is a suite of patches (300K total) that is an attempt to improve Linux security. It offers configuration-free operation, and it gives protection against all kinds of address space modification bugs. It includes a rich ACL system and many auditing systems, and it operates on multiple processor architectures and operating systems.

## 1 Introduction

In our paper, overall, we discuss the theories behind SELinux and grsecurity and compare and contrast the two systems.

“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]” This model is adequate in allowing for enforcement of good security policies. However, traditionally, operating system developers have implemented MAC for their security policy systems in such a way as to disallow users from being able to customize the policy at all, let alone easily. The developers of SELinux designed it in such a way as to make use of the MAC Model of an operating system security policy and, at the same time, overcome the known

limitations of traditional security policy systems using the MAC Model. The security policy of SELinux is meant to be flexible, easily customizable by users.

An Access Control List (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. Grsecurity makes use of ACLs. What sets it apart from other ACL implementations are the definitions it gives to objects, subjects, and rights. It also comes with a tool, *gradm*, which a user can use to fine-tune ACL's for specific processes. Grsecurity includes many security mechanisms besides access control lists, such as protection against buffer overflow exploits, file system protection, auditing, and randomization options.

In section 2, we explain in more detail the MAC Model. Section 3 discusses access control lists. In section 4, we discuss SELinux's Flask architecture, which incorporates the MAC Model and provides for a flexible security policy configuration. The security mechanisms of grsecurity are described in Section 5. In section 6, we discuss our theoretical comparison of SELinux and grsecurity. In section 7, we discuss our results of running benchmark suites on SELinux and grsecurity to provide evidence for any physical differences between the two. 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 Mandatory Access Control

According to [1], mandatory access control is an access control mechanism in which a system controls access to an object and a user cannot change that access. Typically, the system mechanism checks information associated with both the subject and object to decide if the subject may obtain access to the object. Neither the subject nor the object alone determines if access will be granted.

Typically, a MAC mechanism secures information by assigning security levels to information and security clearances to users. The data's security level indicates its sensitivity. All users have access to only the data for which they have a clearance [7].

With MAC, policy rules describe the circumstances in which access is allowed, and the operating system imposes these rules. Policies can be written to restrict access in many ways, such as limiting access to objects based on the time of day. Frequently, users can write to a higher classification, read from a lower classification, and read and write objects of the same classification. Administrators, not resource owners, generally set and change the resource's security level [7].

MAC is suitable for extremely secure systems, such as multilevel secure military applications and mission critical data applications.

## 3 Access Control Lists

Access control lists (ACLs) are a variant of the access control matrix, in which each object is associated with a set of pairs. Each pair contains a subject and a set of rights. Each subject can access the object using any of the rights in the set [1]. This section will discuss typical properties

of access control list implementations, and will explain the implementations in UNIX, some UNIX variants, and Windows NT in detail.

### **3.1 Typical Properties**

In systems that use access control lists, the creation of an object also causes creation of an ACL with some default value. Typically, the creator is initially given all rights, and the subject with own rights is allowed to alter the ACL.

The application of ACLs to privileged users (root or administrator) varies between implementations. For example, Solaris UNIX systems may use abbreviated or full ACLs (see below). In this implementation, if root is the subject, abbreviated ACLs are ignored, but full ACLs apply. Generally, ACLs are applied to privileged users in a limited way.

Some ACL systems support groups and/or wildcards. This helps to decrease the size of the ACL. Additionally, this makes list management more straightforward.

Sometimes, two ACL entries may give conflicting permissions to the same subject. Handling of such conflicts is different between implementations. In AIX, for example, if any entry in an ACL denies access, then access is denied to the subject. Cisco routers, however, apply the first matching ACL entry that matches an incoming packet.

### **3.2 UNIX**

The UNIX operating system uses access control lists in an abbreviated form. UNIX users are divided into three classes, the owner of the file, the group of the file owner, and all other users. Permissions are represented as three triplets - owner, group, and other rights, respectively. Within each triplet, *r*, *w*, and *x*, indicate that read, write, and execute rights are allowed, and - indicates that a right is not allowed.

Access control list abbreviations like these have both good and bad aspects. In this scheme, permissions can be represented by only nine bits. On the other hand, there is a loss of granularity. Because there are only three groups, it is sometimes necessary to compromise (and give a user more or less rights than you desire) or to create new groups (which is burdensome and can only be done by an administrator).

### **3.3 UNIX variants**

Some versions of the UNIX operating system, such as IBM's AIX [1], FreeBSD, IRIX, and Solaris [5], add an access control list to the traditional UNIX user/group/other permissions. These extended permissions allow permissions to be added or deleted from a specific user. For instance, the following represents the permissions for the file *xyzy* in an AIX system [1]:

```
attributes:
base permissions
  owner (bishop) :      rw-
  group (sys) :        r--
  others:              ---
extended permissions enabled
  specify           rw-   u:holly
  permit            -w-   u:heidi, g=sys
```

```

permit          rw-   u:matt
deny           -w-   u:holly, g=facility

```

### 3.4 Windows NT

Windows NT supplies access control lists for files on NTFS partitions. Permissions can be given to read, write, execute, delete, change the permissions of, or take ownership of a file or directory. Windows NT groups these rights into sets called generic rights, as seen in Table 1.

#### Generic rights for files

no access	The subject cannot access the file
Read	The subject can read or execute the file
change	The subject can read, execute, write, or delete the file
full control	The subject has all rights to the file

#### Generic rights for directories

no access	The subject cannot access the directory
Read	The subject can read or execute files within the directory
List	The subject can list the contents of the directory and may change a subdirectory within that directory
Add	The subject may create files or subdirectories in the directory
add and read	The subject may create files or subdirectories in the directory and may read or execute files within the directory
change	The subject can create, read, execute, or write files within the directory and can delete subdirectories
full control	The subject has all rights over the files and subdirectories win the directory

*Table 1: Generic rights in Windows NT*

A file access in Windows NT happens as follows: Windows NT examines the file’s ACL. Access is denied if the user is not present in the ACL and if the user is not a member of any group listed in the ACL or if any ACL entry denies the user access. If access is not denied, the user has the union of the set of rights from each ACL entry that names her [1].

## 4 MAC Model Implementation in SELinux

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

### 4.1 Theory

Traditional MAC mechanisms enforce a policy that 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. [8]”

SELinux makes use of the Flask architecture, which provides flexible support for policy configuration by enhancing the traditional MAC model. 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). 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.

Flask encapsulates security labels (Figure 1), supports flexibility in labeling and access decisions (Figures 2 and 3), and supports policy changes.

```
int security_transition_sid(
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
    security_id_t *out_sid);

ret = security_transition_sid(
    current->sid,
    dir->i_sid,
    SECCCLASS_FILE,
    &sid);
```

**Figure 1: Interface and example call to obtain a security label. The input parameters are the subject SID, the SID of a related object (e.g. the parent directory), and the class of the new object. The SID for the new object is returned as an output parameter [8].**

```
int security_compute_av(
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
    access_vector_t requested,
    access_vector_t *allowed,
    access_vector_t *decided,
    __u32 *seqno);
```

**Figure 2: Interface for obtaining access decisions from the security server. The input parameters are a pair of SIDs, the class of the object, and the set of requested permissions. The pair of SIDs may be subject-to-object, subject-to-subject, or even object-to-object. The granted permissions are returned as output parameters [8].**

```
extern inline
int avc_has_perm_ref(
    security_id_t ssid,
    security_id_t tsid,
    security_class_t tclass,
    access_vector_t requested,
    acv_entry_ref_t *aeref);

ret = avc_has_perm_ref(
    current->sid,
    sk->sid, sk->sclass,
    SOCKET_BIND,
    &sk->avcr);
```

**Figure 3: AVC interface and example call to check permissions. The input parameters are the same as for security\_compute\_av, except for the additional aeref parameter. On its first use, the aeref parameter is set to refer to the AVC entry used for the permission check, and on subsequent checks this reference is used to optimize the lookup. The reference is revalidated on each use to ensure its correctness [8].**

## 4.2 Security Mechanisms

SELinux provides several security mechanisms including ones for process control, file control, and socket control.

Flask has a process management component. Table 2 shows the permissions defined by this component and a brief explanation of each type of permission.

PERMISSION(S)	DESCRIPTION
execute	Execute
transition	Change label
entrypoint	Enter via program
sigkill sigstop sigchld signal	Signal
Fork	Fork
ptrace	Trace
getsched	Get schedule info
setsched	Set schedule info
getsession	Get session
getpgid	Get process group
setpgid	Set process group
getcap	Get capabilities
setcap	Set capabilities

*Table 2: Permissions for the process object class [8].*

The Flask file control mechanism is described in Table 3.

PERMISSION(S)	DESCRIPTION
create	Create
getattr	Get attributes
setattr	Set attributes
inherit	Inherit across execve
receive	Receive via IPC

*Table 3: Permissions for the open file description object class [8].*

Flask provides a mechanism for controlling access to whole file systems (Table 4), individual files (Table 5), and directories (Table 6).

PERMISSION(S)	DESCRIPTION
mount	Mount
remount	Change options
unmount	Unmount
getattr	Get attributes
relabelfrom relabelto transition	Relabel
associate	Associate file

*Table 4: Permissions for the file system object class [8].*

PERMISSION(S)	DESCRIPTION
Read	Read
Write	Write or append
append	Append
Poll	Poll/select
Ioctl	IO control
create	Create
execute	Execute
access	Check accessibility
getattr	Get attributes
setattr	Set attributes
unlink	Remove hard link
Link	Create hard link
rename	Rename hard link
Lock	Lock or unlock
relabelfrom relabelto transition	Relabel

*Table 5: Permissions for the pipe and file object classes [8].*

PERMISSION(S)	DESCRIPTION
add_name	Add a name
remove_name	Remove a name
reparent	Change parent directory
search	Search
Rmdir	Remove
mounton mountassociate	Use as mount point

*Table 6: Additional permissions for the directory object class [8].*

In SELinux, sockets are accessed through file descriptions and therefore inherit permissions defined for the file object classes (Tables 7, 8, and 9).

PERMISSION(S)	DESCRIPTION
Bind	Bind name
Name_bind	Use port or file
connect	Initiate connection
getopt	Get socket options
setopt	Set socket options
shutdown	Shut down connection
recvfrom	Receive from socket
sendto	Send to socket
recv_msg	Receive message
send_msg	Send message

*Table 7: Additional permissions for the socket object classes [8].*

PERMISSION(S)	DESCRIPTION
Listen	Listen for connections
accept	Accept a connection
newconn	Create new socket for connection
connectto	Connect to server socket
acceptfrom	Accept connection from client socket

*Table 8: Additional permissions for the TCP and Unix stream socket object classes [8].*

PERMISSION(S)	DESCRIPTION
getattr	Get attributes
setattr	Set attributes
tcp_recv	Receive TCP packet
tcp_send	Send TCP packet
udp_recv	Receive UDP packet
udp_send	Send UDP packet
Rawip_recv	Receive Raw IP packet
Rawip_send	Send Raw IP packet

*Table 9: Permissions for the network interface and node object classes [8].*

Lastly, SELinux adds to the original Linux application programming interface (API). The enhanced API provides a set of new calls that allow applications to be aware of the new security features. These calls can get SIDS of files, file systems, sockets, and network messages. The robust API makes policy configuration enforcement easier and more understandable.

### 4.3 Policy Configuration

The files indicated in the following table are shared between the kernel and the policy configuration. These are called the Flask definitions. They are meant not to be changed.

Filename	Description
security_classes	Declares the security classes.
initial_sids	Declares initial SIDs.
access_vectors	Defines the access vector permissions for each class.

*Table 10: Flask definition files [9]*

With SELinux one can implement a security model as a combination of a Type Enforcement (TE) model and a Role-Based Access Control (RBAC) model, as described by [8]. 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.

The grammar for the policy language provided by SELinux is contained in `module/checkpolicy/policy_parse.y`. Explanations of the grammar are given in [8]. The original configuration files are all located under the `policy/` directory. The steps to writing one's own security policy with SELinux are as follows:

1. Write flask definitions.
2. Write TE and RBAC declarations and rules.
3. Write user declarations.
4. Write constraint definitions.
5. Write security context specifications.

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 we create. RBAC statements are role declarations, role dominance definitions, or role allow rules. They describe rules for user roles that we 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

```

[9]

The following is a grammar used to specify constraint definitions and constraints on permissions in the form of Boolean expressions.

```

opt_constraints -> constraints | empty
constraints -> constraint_def | constraints constraint_def
constraint_def -> CONSTRAIN classes permissions cexpr ';'
classes -> set
permissions -> set
cexpr -> '(' cexpr ')' | not cexpr | expr and expr | expr or expr |
        U1 op U2 | U1 op user_set | U2 op user_set |
        R1 role_op R2 | R1 op role_set | R2 op role_set |
        T1 op T2 | T1 op type_set | T2 op type_set
not -> '!' | NOT
and -> '&&' | AND
or -> '||' | OR
op -> '==' | '!='
role_op -> op | DOM | DOMBY | INCOMP
user_set -> set
role_set -> set
type_set -> set
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{'
        identifier_list '}'

```

The grammar for the security contexts is as follows:

```

initial_sid_contexts -> initial_sid_context_def |
                    initial_sid_contexts initial_sid_context_def
initial_sid_context_def -> SID identifier security_context
security_context -> user ':' role ':' type
user -> identifier
role -> identifier
type -> identifier

```

[9]

## 5 Grsecurity Overview

Grsecurity is a suite of patches (300K total) that is an attempt to improve Linux security. According to Brad Spengler [11], 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. Keeping systems secure is 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.

## 5.1 ACL Implementation in Grsecurity

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. 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>
  }
}
```

Tables 11 and 12 describe the optional subject and object modes, respectively.

Mode	Description
<b>H</b>	Process is hidden and only viewable by processes with the v mode
<b>V</b>	Process can view hidden processes
<b>P</b>	Process is protected and can only be killed by processes with the k mode or by processes within the same subject.
<b>K</b>	Process can kill protected processes
<b>L</b>	Enables learning mode for this process
<b>D</b>	Protect the /proc/<pid>/fd and /proc/<pid>/mem entries for processes in this subject
<b>B</b>	Enable process accounting for processes in this subject
<b>P</b>	Disables the PAGEEXEC feature of PaX on this subject
<b>S</b>	Disables the SEGMEEXEC feature of PaX on this subject
<b>M</b>	Disables the MPROTECT feature of PaX on this subject
<b>R</b>	Disables the RANDMMAP feature of PaX on this subject
<b>G</b>	Enables the EMUTRAMP feature of PaX on this subject
<b>X</b>	Enables the RANDEXEC feature of PaX on this subject
<b>O</b>	Override the additional mmap() and ptrace() restrictions for this subject
<b>A</b>	Protected the shared memory of this subject. No other processes but processes contained within this subject may access the shared memory of this subject.
<b>K</b>	When processes belonging to this subject generate an alert, kill the process
<b>C</b>	When processes belonging to this subject generate an alert, kill the process and all processes belonging to the IP of the attacker (if there was an IP attached to the process)
<b>T</b>	Ensures this process can never execute any trojaned code
<b>O</b>	Override ACL inheritance for this process

*Table 11: Subject Modes*

Mode	Description
<b>R</b>	Object can be opened for reading
<b>W</b>	This object can be opened for writing or appending
<b>X</b>	This object can be executed (or mmap'd with PROT_EXEC into a task)

Mode	Description
A	This object can be opened for appending
H	This object is hidden
T	This object can be ptraced, but cannot modify the running task. This is called 'read-only ptrace'
S	Logs will be suppressed for denied access to this object
I	When the object is executed, it inherits the ACL of the subject in which it was contained. This mode only applies to binaries.
R	Audit successful reads to this object
W	Audit successful writes to this object
X	Audit successful execs of this object
A	Audit successful appends to this object
F	Audit successful finds of this object
I	Audit successful ACL inherits of this object

*Table 12: Object Modes*

It is also possible to define the following restrictions on system resources:

RES_CPU	CPU time in milliseconds
RES_FSIZE	Maximum file size in bytes
RES_DATA	Maximum data size in bytes
RES_STACK	Maximum stack size in bytes
RES_CORE	Maximum core size in bytes
RES_RSS	Maximum resident set size in bytes
RES_NPROC	Maximum number of processes
RES_NOFILE	Maximum number of open files
RES_MEMLOCK	Maximum locked-in-memory in bytes
RES_AS	Address space limit in bytes
RES_LOCKS	Maximum file locks

*Table 13: Restrictions on System Resources*

For every event, the kernel will check the ACLs for the executing process and the requested object. If both ACLs allow the event, it will be executed.

Inheritance is provided to reduce the necessary configuration needed for similar binaries. Given a parent and child ACL, if an object in the parent does not exist in the child, the object from the parent is added to the child ACL. For example,

An ACL like:	Would expand to:
/ {	/ {
/                  rwx	/                  rwx
/etc              rx	/etc              rx
/usr/bin          rx	/usr/bin          rx
/tmp              rw	/tmp              rw
}	}
/usr/bin/mailman {	/usr/bin/mailman {
/tmp              rwx	/                  rwx
}	/etc              rx
	/usr/bin          rx
	/tmp              rwx
	}

Inheritance is computed for all parents of the path, and is applied to all ACLs, unless “o” is found in the subject mode [12].

This implementation of ACLs implements 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.

### 5.1.1 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>
}
```

### 5.1.2 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. Additionally, learning mode supports files, capabilities, resources, and socket usage.

The steps for using learning mode on a new process are:

1. Enable the ACL system
2. Build a default restrictive ACL for the new process. Add “l” to the subject mode of this ACL. This will disable the ACL system for this process only. When running the process, every access attempt that would be denied by the default ACL will be logged, but the accesses will be allowed to occur.
3. Run the process several times, at least 4. Use of the process should be as thorough and as close to real life as possible, since learning mode uses a threshold-based system.
4. Disable the ACL system
5. Run gradm -L -O /etc/grsec/acl to place the new learned ACLs at the end of your ruleset
6. Remove the old obsolete ACLs.

Thus, learning mode can be used to create an access control list that is optimized for a new process on a particular environment.

## 5.2 Additional Security Mechanisms

Grsecurity includes many security mechanisms besides access control lists, such as protection against buffer overflow exploits, file system protection, auditing, and randomization options. This section will explore those mechanisms.

## 5.2.1 Address Space Protection

Many Linux exploits take advantage of how Linux handles memory. PaX address space protection attempts to deal with this problem by creating defense mechanisms against exploits that give an attacker access to the attacked task's address space [2].

Linux by default does not understand executable pages; all readable pages are executable. Consequently, exploits can inject executable code in memory regions that shouldn't be executable [4]. PaX, however, implements non-executable virtual memory pages with its `SEGMEXEC` option.

Two implementations are available for enforcing non-executable pages. First, *paging based non-executable pages* is based on the CPU's paging features. Second, *segmentation based non-executable pages* splits the address space into data and code segments. While the first option may have high performance impacts on some architectures, the second option limits the address space of applications. Administrators can test applications to choose the best implementation [4].

Additionally, PaX provides full address space layout randomization (ASLR) for ELF binaries. This feature was created because most attacks require some advance knowledge of assorted addresses in the attacked task. PaX attempts to introduce randomization in these addresses each time a task is created. Thus, an attacker is forced to guess each address or attain it by brute force [2]. With this option activated, exploits will probably crash the attacked application, making it easy to catch and react to the attack [4]. The available ASLR options are:

- § *Randomize kernel stack base.* Randomize every task's kernel stack on every system call.
- § *Randomize mmap() base.* Randomizes the location of dynamically loaded libraries
- § *Randomize user stack base.* Randomizes every task's userland stack.

Additional PaX options [4] are:

- § *Emulate trampolines.* Some tools need to execute small snippets of code from a non-executable memory page (which is not possible with the page and segmentation based non-executable pages options). This option allows the user to execute those programs, while still using non-executable pages.
- § *Restrict mprotect().* Do not allow programs to switch between several possible page protection modes.
- § *Disallow ELF text relocations.* Force your system to use position independent code ELF libraries only.
- § *Deny writing to /dev/mem and /dev/port.* Make it harder for attackers to insert malicious code into the running kernel.
- § *Disable privileged I/O.* Disable the possibility for tools to modify the running kernel.
- § *Remove addresses from /proc/pid/maps.* Should be used if any of the randomization algorithms are used.
- § *Hide kernel symbols.* Control who can know about loaded modules and kernel symbols.

As described earlier, the `grsecurity` ACL system supports many PaX flags. These flags (as described with more detail) are described below.

<b>P</b>	Disables the PAGEEXEC feature of PaX on this subject. PAGEEXEC implements non-executable pages using the paging logic of IA-32 based CPUs [2].
<b>S</b>	Disables the SEGMEXEC feature of PaX on this subject. SEGMEXEC implements non-executable paging based on the segmentation logic of IA-32 based CPUs [2].
<b>M</b>	Disables the MPROTECT feature of PaX on this subject. This helps to prevent the introduction of new executable code into a task's address space. This feature includes the mmap and mprotect restrictions mentioned above [2].
<b>R</b>	Disables the RANDMMAP feature of PaX on this subject. RANDMMAP introduces randomness into memory regions handled by the do_mmap() kernel interface. Included are all file mappings, and all mappings of the main executable, libraries, the brk(), and mmap() managed heaps [2].
<b>G</b>	Enables the EMUTRAMP feature of PaX on this subject. EMUTRAMP emulates certain instruction sequences that are known to be generated at runtime in otherwise non-executable memory regions. This is needed when non-executable pages are enforced with PAGEEXEC or SEGMEXEC, since trying to execute the instruction sequences would cause the task to terminate. Only selected types of runtime generated code are supported [2].
<b>X</b>	Enables the RANDEXEC feature of PaX on this subject. RANDEXEC introduces randomness into the main executable file mapping addresses [2].

*Table 14: PaX Flags*

By default, PAGEEXEC, SEGMEXEC, MPROTECT, and RANDMAP are all enabled on ELF binaries in the system [12].

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. Despite this difficulty, this feature of grsecurity is very important, as it prevents stack overflow exploits by making the stack non-executable.

## 5.2.2 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 [6]. Sometimes, it is subject to exploits, so grsecurity provides *Proc restrictions*. While using Proc restrictions, 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 [4].

Furthermore, filesystem protection is available using

- § *Linking restrictions*. Hardlinking to files that you do not own is not possible. Users can only follow symbolic links in the world-writable directories if they are the owner of that directory.
- § *FIFO restrictions*. Users can only write to FIFOs in world-writable *+t* directories if they are the owner of the FIFO or directory.
- § *Chroot jail restrictions*. It is usually easy for a root user in chroot to break out. Grsecurity strengthens it by making syscalls unrelated to the filesystem aware of chroot, enforcing `chdir("/")` upon chroot, and lowering the capabilities upon chroot [11].

## 5.2.3 Kernel Auditing

Grsecurity allows administrators to configure the amount of logging provided by the kernel [4]. This auditing capability is meant to detect attacks. Audited events include [11]:

- § Exec
- § Chdir(2)

- § Mount(2)
- § Unmount(2)
- § IPC creation and deletion
- § Signals, such as SIGSEGV, SIGABRT, SIGBUS, SIGILL
- § Failed forks
- § Ptrace(2)
- § Time changes using stime(2) and settimeofday(2)
- § Execs inside chroot(2)
- § Denied capabilities

### 5.2.4 Executable Protections

Executable protection is provided in grsecurity, since most exploits work through or with running processes. Some options for executable protection are [4]:

- § *Enforce RLIMIT\_NPROC.* Check resource limits on processes during execve() calls.
- § *Dmesg(8) restriction.* Do not allow non-users to use dmesg() to view the log buffer.
- § *Randomized PIDs.* Force attackers to guess the process IDs. This prevents filesystem races and adds randomness to programs that use getpid(2) and srandom(3) seeding [11].

### 5.2.5 Network Protections

The default Linux TCP/IP stack implementation is vulnerable to prediction-based attacks. Grsecurity contains many options to make these predictions much more difficult. These options include [4]:

- § *Randomized IP IDs.* Prevents operating system fingerprinting and spoofed scans [11].
- § *Randomized RPC XIDs.* Randomizes the RPC transaction IDs (XIDs) for RPC requests to prevent RPC connection hijacking [11].
- § *Larger entropy pools.* Doubles the pool size
- § *Truly random TCP ISN selection.* Randomize the TCP initial sequence numbers.
- § *Randomized TCP source ports.* Randomize the dynamically generated connect() source port.
- § *Altered Ping IDs.* Alter the ICMP Echo Replies to make their IDs equal to the ID of the ICMP Echo Requests they respond to.
- § *Socket restrictions.* Select a GID on the system in which members will have restricted socket access. Options for restrictions include *Deny any sockets to group*, *Deny client sockets to group*, and *Deny server sockets to group*.
- § *Trusted path execution.* Allow users in the mentioned GID to execute binaries from root-owned, non-writeable directories.
- § *Partially restrict non-root users.* Users that are not in the mentioned GID are only allowed to execute files in the directories they own and that are not (1) group/world writeable, or (2) unwritable and owned by root.

### 5.2.6 Fork bomb protection

Grsecurity offers some protection against “fork bombs.” These are programs that run extra copies of themselves so that resources are denied to other system users [3].

### 5.2.7 Sysctl Support

With sysctl support turned on, it is possible to active or deactivate grsecurity options on the fly on a running system.

## 6 MAC/ACL Model and Implementation Comparison

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

### 6.1 Protection Models

Grsecurity has a simple MAC implementation. It does not implement the domain-type enforcement component of SELinux, though the grsecurity ACL 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. 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]). On the other hand, grsecurity supports certain features that SELinux lacks. For example, grsecurity supports fork() rate limiting, various network limiting and randomizing options, and randomized PIDs. Grsecurity also includes PaX for protection against stack smashing attacks and the like.

Both grsecurity and SELinux will respect the DAC if the DAC permits less access than the MAC model.

### 6.2 Installation

Grsecurity is much simpler to install than SELinux. To install grsecurity, the site administrator need only patch the kernel with the gresecurity 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 also 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.

### 6.3 Ease of Use

Overall, grsecurity is simpler to administer. 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 tweak these policies somewhat, however.

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 tweak 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, since the goal of this script is not generate secure policies. 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 relabel all the files associated with the context. Gentoo and Debian both provide package managers that support for automatic file relabeling. 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.

## **6.4 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 ACLs. Additional information can be found in forums and in informal web sites. We found that information about even simple things (such as the sequence of commands to use to reload a new policy) was difficult to find.

## **6.5 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 [10].

Basically, the LSM kernel patch provides a framework to support access control modules. Alone, the framework doesn't 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 unregistering 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 [10].

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 hood 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 [13].” 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.

In an IRC conversation, Russell Coker, a Red Hat Linux employee and SELinux expert, stated that he was not overly concerned with rootkit possibility. He stated that most of the information provided by the LSM hooks can already be gathered by intercepting the system call table. He felt that the maintenance benefits of having all the linux security patches use the same core outweighed the fact that rootkits would be “marginally easier” to develop.

These beliefs about Linux Security Modules represent fundamental differences in the design of the two systems.

## 7 System Performance Expectations and Benchmarking

As we have discussed, SELinux and grsecurity differ slightly in their respective security models, and those differences propagate further as each model and its policies are implemented and enforced. While this makes for interesting study and debate, we wish to quantify how these two approaches and subsequent implementations differ in terms of performance. As in [14], the best way to evaluate performance is through objective benchmarks to measure performance. In order to provide a statistical basis of comparison for the two patches, we chose to replicate the benchmarking tests conducted in [8] not to compare performance penalties and overhead against a baseline operating systems installation, but rather to extend the benchmarking tests to grsecurity and compare performance differential relative to SELinux.

Microbenchmarking suites such as Lmbench and Unixbench perform a sequence of low-level operations in a controlled, timed environment and report understandable metrics in terms of performance. Lmbench reports program timings in microseconds, therefore lower results indicate shorter running times, and hence, better performance. Unixbench, on the other hand, aggregates program performance into a set of index scores, where a higher score indicates more execution loops in a fixed time, therefore better performance. The two benchmarking suites both employ a number of sub-programs, or microbenchmarks, they execute a sequence, repetition, or combination thereof of low-level, sometimes atomic operations. The low level of these tests provides the transparency and precision required in order to make informed conclusions regarding performance. Rather than timing high-level user programs and simply reporting the results, we chose to emulate and extend work already done on SELinux in order to come to a greater understanding of how both implementations impact systems performance and compare side-by-side.

We established a testbed in two identical PC's, each configured with Gentoo Linux 1.4.1, running on 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 in three modes – *root*, *user*, and *enforcing* mode where security policies are implemented.

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 a relatively high-level view, our hypothesis contends that there will be no significant overall difference in observed system performance between SELinux and grsecurity, but that there will 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. The results themselves cannot be argued since our experiments were conducted in a controlled environment with identical equipment and kernel, with the only difference being the security patch applied to each.

## **7.1 Unixbench**

Our results for Unixbench are shown in Table 15 and highlighted graphically in Figure 4. The tests we chose to include correlate to most of the tests highlighted in [8], but 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.

Dhrystone 2 executes non-floating point operations that are commonly found in user programs. Whetstone performs arithmetic operations on double-precision floating point variables. Exec1 replaces a currently running process with a new one. The file copy operations capture the number of characters copied to a file based on the various buffer sizes indicated. Pipe throughput 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. The shell script test measures the execution of a shell script by 8 concurrently running processes. Finally, syscall evaluates the time required to execute a fixed sequence of system calls.

In the tests consisting of arithmetic, user program and file copy operations, we observe fairly consistent performance between the two patches when security is not enabled. Due to more restrictive policy enforcement in grsecurity, however, we note distinct overhead in these tests, except in the case of arithmetic operation. Since arithmetic operations are atomic in nature and are executed on-chip with no external file or process operations, there is no performance penalty as a result of any security mechanism. We observe significant performance penalties in SELinux for pipe throughput and pipe-based context-switching as a result of the revalidation of permissions required even when security is not enabled. Since subject-object associations must still be verified in this mode, this results in additional overhead relative to grsecurity. Unlike in [8], we observed no significant overhead resulting from process creation and termination in SELinux. There is improved shell script execution in both patches when users execute this script as opposed to root.

	<u>Root</u>		<u>User</u>		<u>Root-Enforcing Mode</u>	
	<u>SELinux</u>	<u>grsecurity</u>	<u>SELinux</u>	<u>grsecurity</u>	<u>SELinux</u>	<u>grsecurity</u>
Dhrystone 2	69.4	69.5	69.4	69.2	69.4	69.5
Whetstone	35.4	35.4	35.4	35.4	35.4	35.4
Execl	155.6	163.6	147.9	165.6	172.9	164.2
File Copy 1K	71.6	79.3	72.5	77.5	73.6	78.1
File Copy 256	83.3	95.8	84.8	94.7	87.3	95.7
File Copy 4K	50.8	53.1	51.3	53.2	51.8	36.7
Pipe Throughput	134.0	256.7	155.3	268.9	164.6	194.0
Context Switching	162.8	230.3	154.2	257.3	162.4	259.9
Process Creation	159.0	149.5	164.2	156.7	167.5	163.9
Shell Scripts (8)	252.4	266.7	325.8	355.0	267.2	283.9
Syscall	166.5	202.2	178.1	205.5	178.7	203.0

Table 15: Unixbench results

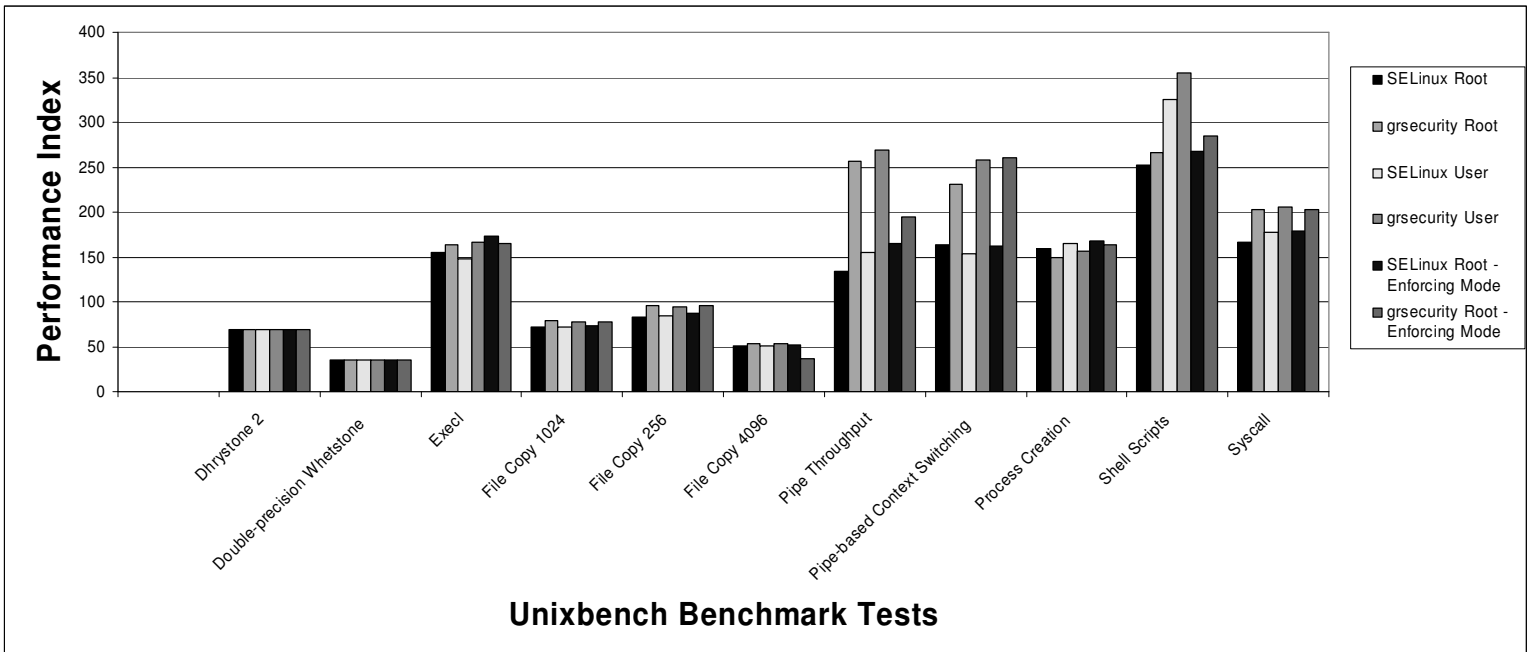


Figure 4: Unixbench results

## 7.2 Lmbench

Our results for Lmbench are shown in Table 16 and highlighted graphically in Figure 5a, 5b and 5c. Three graphs are included due to the wide range of timing results that Lmbench tests report. The tests included here, as in Unixbench, are a subset of all microbenchmark tests available for analysis.

Syscall measures how long it takes to write one byte to /dev/null, therefore providing a baseline for interaction with the kernel. Read and write both time atomic memory operations with four

byte integers. Fstat measures the time required to get file status of an open file. Stat does the same with a file in the file system that is not open. Open/close simply times the opening and closing of a temporary file for reading. Pipe latency, AF\_UNIX, UDP and TCP latency and TCP/IP tests all time inter-process communication between two processes on the tested system. Fork, execve and /bin/sh time process creation from fork() and exit, to fork() and execve, to fork() and instantiating the shell.

We observe pronounced overhead in SELinux in the syscall and memory tests even in unenforced mode, largely as a result of having to revalidate permissions for each operation. This overhead is not manifested in grsecurity. Performance in tests of inter-process communication and process creation reveal similar performance metrics between the two patches, but SELinux appears to perform consistently slower in these tests than grsecurity, although not by an order of magnitude or more, as in the memory and system call tests.

	<u>Root</u>		<u>User</u>		<u>Enforcing Mode - Root</u>	
	<u>SELinux</u>	<u>grsecurity</u>	<u>SELinux</u>	<u>grsecurity</u>	<u>SELinux</u>	<u>grsecurity</u>
Syscall	0.60	0.46	0.60	0.46	0.60	0.46
Read	2.10	0.84	2.14	0.84	2.20	0.95
Write	1.94	0.68	1.98	0.68	1.98	0.74
Stat	16.97	6.79	15.92	6.51	15.59	7.92
Fstat	2.10	1.15	2.05	1.15	2.13	1.20
Open/close	21.85	8.72	19.37	8.46	18.45	10.96
Fork	505.82	483.26	490.64	521.82	483.60	492.36
Execve	1592.83	1579.92	1555.42	1520.58	1533.81	1661.00
/bin/sh	6393.33	6340.33	6576.00	6373.00	6278.33	6529.67
Pipe latency	15.82	9.93	14.41	10.32	15.91	10.69
AF_UNIX	25.39	22.55	24.58	23.89	25.82	23.21
UDP latency	43.53	42.23	47.51	39.86	46.44	43.04
TCP latency	79.93	69.95	79.74	71.07	82.10	69.83
TCP/IP	286.38	283.20	287.10	281.19	281.31	218.49

*Table 16: Lmbench results*

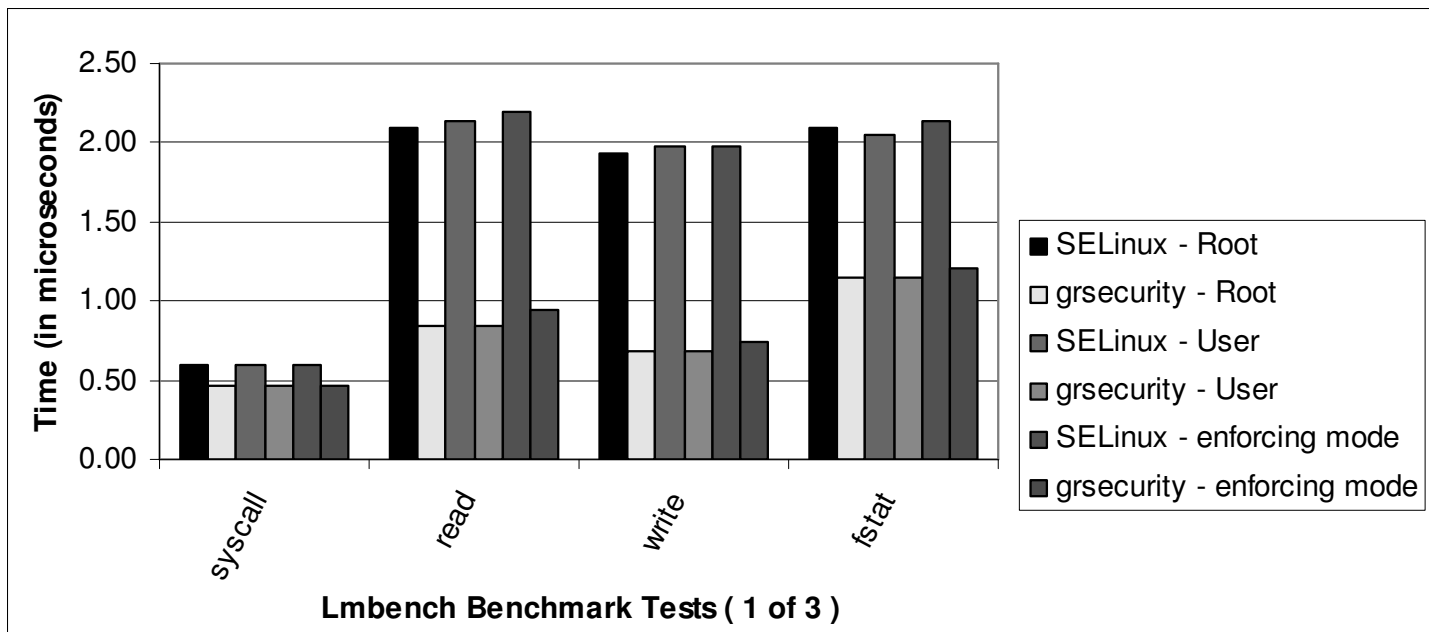


Figure 5a: Lmbench results

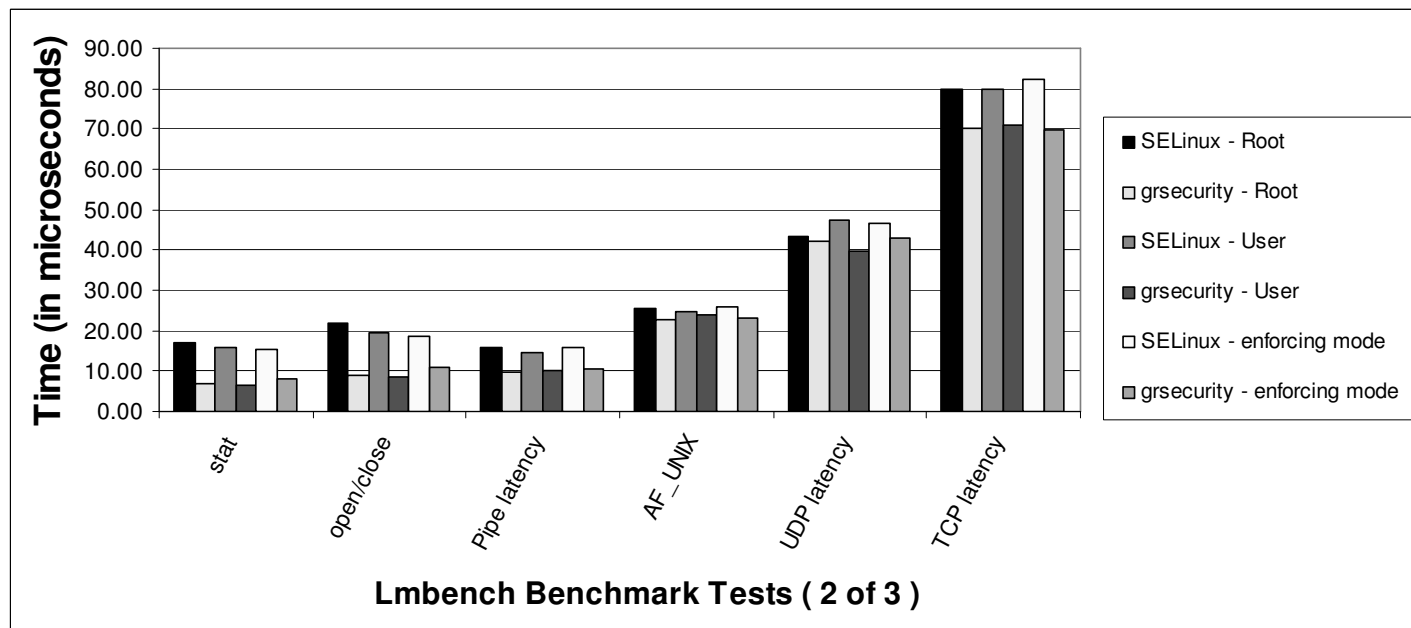


Figure 5b: Lmbench results

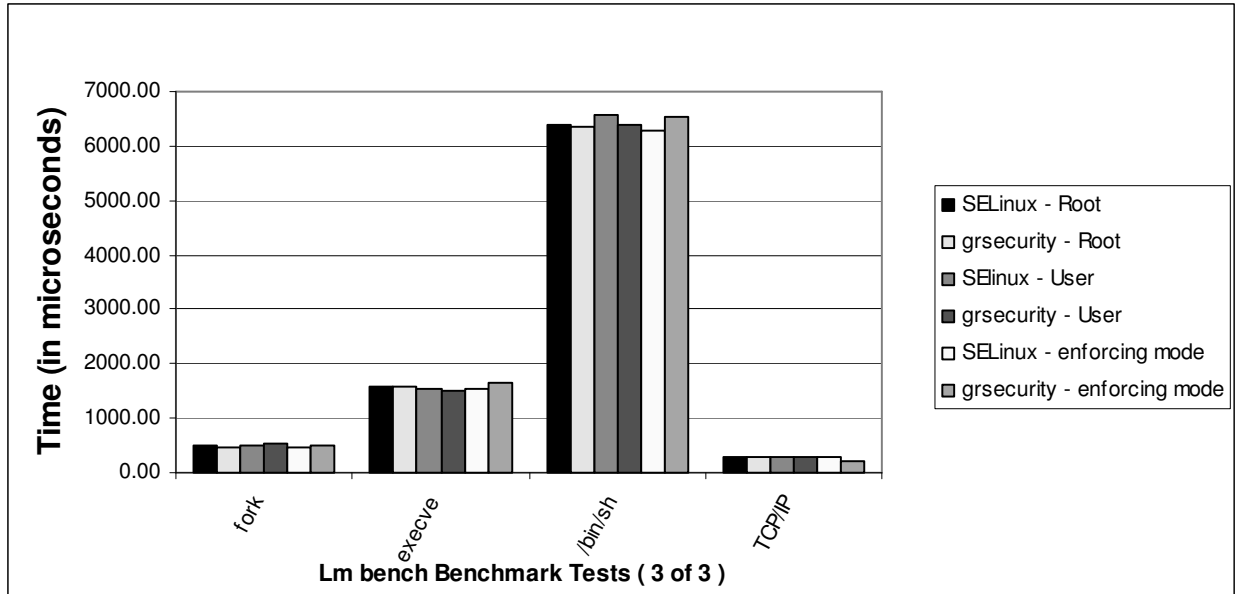


Figure 5c: Lmbench results

### 7.3 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. Unixbench’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 assess each patch’s relative performance in this domain. Arithmetic operations, user program operations and process creation in general all appear to incur little overhead, whereas memory accesses in SELinux result in significant overhead due to revalidation of subject-object permissions. 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 tasks, and hence processes, 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.

## 8 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.

### 8.1 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:

```
/usr/bin/irssi lo {
  / h
  -CAP_ALL
  RES_FSIZE          0 0
  RES_DATA           0 0
  RES_RSS            0 0
  RES_NOFILE         0 0
  RES_MEMLOCK        0 0
  RES_STACK          0 0
  RES_AS             0 0
  RES_NPROC          0 0
  RES_LOCKS          0 0

  connect {
    disabled
  }

  bind {
    disabled
  }
}
```

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.

This policy has been further refined by hand to produce the following policy.

```

/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/*/exe
    /proc/*
    /lib rx
    /lib/ld-2.3.2.so x
    /home/*/.irssi/config r
    /home/*/.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 {
        0.0.0.0:6667 stream tcp
        0.0.0.0:6667 stream tcp
        0.0.0.0:53 dgram udp
    }

    bind {

```

```
    0.0.0.0:0 dgram ip
}
```

This policy has been modified 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.

## 8.2 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..

Below is the policy shown with the macros expanded.

```

#DESC irssi - IRC client
#

type irssi_t, domain, privlog;
type irssi_exec_t, file_type, sysadmfile, exec_type;
role sysadm_r types irssi_t;

# Allow the process to transition to the new domain.
allow sysadm_t irssi_t:process transition;

# Do not audit when glibc secure mode is enabled upon the transition.
dontaudit sysadm_t irssi_t:process noatsecure;

# Allow the process to execute the program.
allow sysadm_t irssi_exec_t:file { read getattr lock execute ioctl };

# Allow the process to reap the new domain.
allow irssi_t sysadm_t:process sigchld;

# Allow the new domain to inherit and use file
# descriptions from the creating process and vice versa.

```

```

allow irssi_t sysadm_t:fd use;
allow sysadm_t irssi_t:fd use;

# Allow the new domain to write back to the old domain via a pipe.
allow irssi_t sysadm_t:fifo_file { ioctl read getattr lock write append };

# Allow the new domain to read and execute the program.
allow irssi_t irssi_exec_t:file { read getattr lock execute ioctl };

# Allow the new domain to be entered via the program.
allow irssi_t irssi_exec_t:file entrypoint;

type_transition sysadm_t irssi_exec_t:process irssi_t;

allow irssi_t { root_t usr_t lib_t etc_t }:dir { read getattr lock search ioctl };
allow irssi_t lib_t:lnk_file { read getattr lock ioctl };
allow irssi_t ld_so_t:file { read getattr lock execute ioctl };
allow irssi_t ld_so_t:file execute_no_trans;
allow irssi_t ld_so_t:lnk_file { read getattr lock ioctl };
allow irssi_t shlib_t:file { read getattr lock execute ioctl };
allow irssi_t shlib_t:lnk_file { read getattr lock ioctl };
allow irssi_t ld_so_cache_t:file { read getattr lock ioctl };
allow irssi_t device_t:dir search;
allow irssi_t null_device_t:chr_file { ioctl read getattr lock write append };

role user_r types irssi_t;
role staff_r types irssi_t;

# Allow the process to transition to the new domain.
allow userdomain irssi_t:process transition;

# Do not audit when glibc secure mode is enabled upon the transition.
dontaudit userdomain irssi_t:process noatsecure;

# Allow the process to execute the program.
allow userdomain irssi_exec_t:file { read getattr lock execute ioctl };

# Allow the process to reap the new domain.
allow irssi_t userdomain:process sigchld;

# Allow the new domain to inherit and use file
# descriptions from the creating process and vice versa.
allow irssi_t userdomain:fd use;
allow userdomain irssi_t:fd use;

# Allow the new domain to write back to the old domain via a pipe.
allow irssi_t userdomain:fifo_file { ioctl read getattr lock write append };

# Allow the new domain to read and execute the program.
allow irssi_t irssi_exec_t:file { read getattr lock execute ioctl };

# Allow the new domain to be entered via the program.
allow irssi_t irssi_exec_t:file entrypoint;
type_transition userdomain irssi_exec_t:process irssi_t;

# Allow the domain to create and use UDP and TCP sockets.
# Other kinds of sockets must be separately authorized for use.
allow irssi_t self:udp_socket { create ioctl read getattr write setattr append bind
connect getopt setopt shutdown };
allow irssi_t self:tcp_socket { create ioctl read getattr write setattr append bind
connect getopt setopt shutdown listen accept };

# Allow the domain to send UDP packets.

```

```

# Since the destination sockets type is unknown, the generic
# any_socket_t type is used as a placeholder.
allow irssi_t any_socket_t:udp_socket sendto;

# Allow the domain to send using any network interface.
# netif_type is a type attribute for all network interface types.
allow irssi_t netif_type:netif { tcp_send udp_send rawip_send };

# Allow packets sent by the domain to be received on any network interface.
allow irssi_t netif_type:netif { tcp_recv udp_recv rawip_recv };

# Allow the domain to receive packets from any network interface.
# netmsg_type is a type attribute for all default message types.
allow irssi_t netmsg_type:{ udp_socket tcp_socket rawip_socket } recvfrom;

# Allow the domain to initiate or accept TCP connections
# on any network interface.
allow irssi_t netmsg_type:tcp_socket { connectto acceptfrom };

# Receive resets from the TCP reset socket.
# The TCP reset socket is labeled with the tcp_socket_t type.
allow irssi_t tcp_socket_t:tcp_socket recvfrom;

dontaudit irssi_t tcp_socket_t:tcp_socket connectto;

# Allow the domain to send to any node.
# node_type is a type attribute for all node types.
allow irssi_t node_type:node { tcp_send udp_send rawip_send };

# Allow packets sent by the domain to be received from any node.
allow irssi_t node_type:node { tcp_recv udp_recv rawip_recv };

# Allow the domain to send NFS client requests via the socket
# created by mount.
allow irssi_t mount_t:udp_socket { ioctl read getattr write setattr append bind
connect getopt setopt shutdown };

# Bind to the default port type.
# Other port types must be separately authorized.
allow irssi_t port_t:udp_socket name_bind;
allow irssi_t port_t:tcp_socket name_bind;

# 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 };

# Allow the process to modify the directory.
allow irssi_t user_home_dir_t:dir { read getattr lock search ioctl add_name
remove_name write };

# Allow the process to create the file.

```

```

allow irssi_t user_home_irssi_t:dir { create read getattr lock setattr link unlink
rename search add_name remove_name reparent write rmdir };

type_transition irssi_t user_home_dir_t:dir user_home_irssi_t;

# Allow the process to modify the directory.
allow irssi_t user_home_t:dir { read getattr lock search ioctl add_name remove_name
write };

# Allow the process to create the file.
allow irssi_t user_home_irssi_t:file { create ioctl read getattr lock write setattr
append link unlink rename };

type_transition irssi_t user_home_t:file user_home_irssi_t;

allow irssi_t user_home_t:file { getattr read write };
allow irssi_t etc_t:file { getattr read };

# locale support
allow irssi_t etc_t:lnk_file read;
allow irssi_t locale_t:dir { read getattr lock search ioctl };
allow irssi_t locale_t:{ file lnk_file } { read getattr lock ioctl };

# 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 { ioctl read getattr lock write append };

# 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 };

```

## 9 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 purposely compared the two in terms of their theory and practicality so as to provide a deeper understanding of how one vies with the other. It is common knowledge that theory and practice are two very different entities and sometimes a system which appears to be more theoretically sound than another is sometimes less practical.

### 9.1 Conclusions Pertaining to Theory

From a theoretical standpoint, SELinux is a more powerful access control mechanism, since it incorporates role-based access control (RBAC). Nevertheless, we believe the two theories allow for sound security models. They both allow for easy control of access between processes and objects, processes and other processes, and objects and other objects.

## **9.2 Conclusions Pertaining to Practice**

The tools and capabilities that come with each set one security system apart from the other. The Flask architecture in SELinux provides for a flexible security policy. This means that the administrator can very easily manipulate and customize the policy by simply modifying a set of policy files written in a policy language set forth by the developers of SELinux. The policy language is not so easy to learn but allows for efficient methods of customization. Based upon our experimental implementation of a security policy using SELinux, we conclude that this policy language is powerful and robust and should be considered (by one choosing to use one security system over the other) as the most dominating advantage it has. Grsecurity, on the other hand, comes with the `gradm` tool, which is capable of programmatically optimizing and fine-tuning ACLs in the operating system. In basing his choice on whether he likes one system over the other, one should first decide whether he wants the flexibility but semi-difficulty that SELinux offers or the somewhat inflexibility but ease that grsecurity offers.

In terms of performance of one system over the other, we conclude that they are generally equal in quality. Although, we noted several small differences of performance in very specific areas, we believe that these differences balance out and do not hold much water in helping one choose his preference of one system over the other.

## **10 Further Work**

In [8], Loscocco and Smalley used two benchmarking suites in order to analyze the performance of SELinux relative to a baseline distribution. Herein, we have extended their work to compare and analyze the performance of SELinux and grsecurity to each other, noting largely insignificant or inconclusive performances differences between the two. As Linux continues to proliferate in industry, government and academia, managers must make informed implementation decisions based on many factors, including such high-level concepts as performance, security, ease of use, and so forth. While many trade publications provide *some* statistical analysis of off-the-shelf operating systems and applications, a thorough, detailed and extensive analysis of several Linux security options beyond just SELinux and grsecurity, in a comparative fashion, would be of great benefit the community. Decision-makers are generally aware that increasing security incurs performance penalties, but having a clear and undeniable understanding of which performance domains are more affected by various security options would allow for better decisions. There are many benchmarking suites available in the public domain to test operating system performance. Since the OS represents the confluence of hardware and software in a system, there is virtually no limit to the variation and permutation of tests that could be conducted.

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

Furthermore, in a future version of this paper, 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 role-based access control. It would also be interesting to compare Rule Set Based Access Control (RSBAC), a less popular MAC system that incorporates both RBAC and DTE. In a future revision, 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] Matt Bishop. Computer Security Art and Science.
- [2] Documentation for the PaX Project. <http://pageexec.virtualave.net/docs/>.
- [3] David Elson. Grsecurity. In *SecurityFocus*, February 28, 2002. <http://www.securityfocus.com/infocus/1551>.
- [4] Gentoo Linux Grsecurity Guide. <http://www.gentoo.org/proj/en/hardened/grsecurity.xml>.
- [5] Daniel Harris. FreeBSD Access Control Lists. [http://www.onlamp.com/pub/a/bsd/2003/08/14/freebsd\\_acls.html](http://www.onlamp.com/pub/a/bsd/2003/08/14/freebsd_acls.html). 8/14/2003.
- [6] Linux / Unix Command: proc. [http://linux.about.com/library/cmd/blcmdl5\\_proc.htm](http://linux.about.com/library/cmd/blcmdl5_proc.htm).
- [7] Mandatory Access Control. <http://www.rz.informatik.uni-muenchen.de/doku/web/sec/v11/ch08s02.html>.
- [8] Peter Loscocco and Stephen 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.
- [9] Stephen Smalley. Configuring the SELinux Policy. Technical Report 02-007, NSA and NAI Labs, February 2002.
- [10] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux Security Modules: General Security Hooks for Linux. <http://lsm.immunix.org/docs/overview/linuxsecuritymodule.html>.
- [11] Brad Spengler. Detection, Prevention, and Containment: A Study of grsecurity. <http://grsecurity.unc.bl.ac.yu/grsecurity-slide.ppt>.
- [12] Brad Spengler. Grsecurity ACL Documentation V1.5, April 1, 2003. <http://www.grsecurity.org/gracldoc.pdf>.
- [13] Brad Spengler. LSM. <http://www.grsecurity.net/lsm.php>.
- [14] Rodney C. Wilson. Unix Test Tools and Benchmarks. Prentice-Hall. Upper Saddle River, NJ 1995. 152-155.