**Flexible Mandatory Access Control (MAC) in Modern Operating Systems**

Jeffrey H. Jewell

The University of Colorado at Colorado Springs

Colorado Springs, CO U.S.A

jjewell@uccs.edu

*Abstract*— In mainstream computing environments there has always existed a flawed assumption that security can be adequately provided in application space without the additional security features inherent in the underlying operating system. It seems the computer industry has not accepted the critical role of the operating system to security as can be seen in all the exploits and patches that result with new releases [2]. In this paper we examine the use of mandatory access control mechanisms in the Linux platform and comment on their effectiveness in meeting security objectives.

# Introduction

The common control mechanism used in today’s modern operating systems is called Discretionary Access Control (DAC). The Trusted Computer System Evaluation Criteria (TCSEC) defines DAC as "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong [7]. The controls are [discretionary](http://en.wiktionary.org/wiki/discretionary) in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. So individual users are given the ability to make security policy decisions and/or assign security attributes. In the UNIX world this equates with the concept of user, group, and execution privileges assigned to a user.

DAC has many disadvantages when it comes to security.

* First it makes its decisions based solely on user identity and ownership. It does not consider other factors such as role of the user, the function and trustworthiness of the program, and sensitivity and integrity of the data. So it fails to meet 2 of the basic security services (CIA) – confidentiality and integrity.
* A program run by a user inherits all the permissions associated with the program. The user is free to change permissions on other objects.
* Privileged programs run with coarse grain control so that if this program is exploited it can give the user complete access to the system.

End systems must be able to enforce the separation of information based on confidentiality and integrity requirements to provide system security. Operating system security mechanisms are the foundation for ensuring such separation. Unfortunately, existing mainstream operating systems lack the critical security feature required for enforcing separation: mandatory access control. As a consequence, application security mechanisms are vulnerable to tampering and bypass, and malicious or flawed applications can easily cause failures in system security.

Adding mandatory access control mechanisms to the operating system can help mitigate some of vulnerabilities associated with DAC. The term mandatory access control (MAC) refers to a type of [access control](http://en.wikipedia.org/wiki/Access_control) by which the [operating system](http://en.wikipedia.org/wiki/Operating_system) constrains the ability of a *subject* or *initiator* to access or generally perform some sort of operation on an *object* or *target* [6]. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system [kernel](http://en.wikipedia.org/wiki/Kernel_(computing)) examines these security attributes and decides whether the access can take place. Any operation by any subject on any object will be tested against the set of authorization rules (aka *policy*) to determine if the operation is allowed.

Traditional MAC mechanisms in use today have tended to be too tightly coupled to the DOD multi-level security policy for mandatory security, however, that policy is too restrictive to meet the various security policies required by today’s systems [3][4]. Access decisions in MLS are based on clearances for subjects and classifications for objects. This provides a very static lattice that allows the system to decide by a subject’s clearance level which objects can be and written to. The focus of MLS is entirely on maintaining confidentiality. It only ensures that unauthorized users cannot read from or write to certain files.

*Organization:* The rest of the paper is organized as follows: In the next section, we review some of the previous work done by the NSA to provide a more flexible general MAC architecture that addresses integrity. In Section III we examine how this new MAC architecture found its way into the Linux operating system. In Section IV we look at a specific implementation of the architecture called the Security Enhanced Linux operating system and summarize our research into some of the more relevant features available to a system administrator or policy implementer. In Section V we show live screenshots of some of the features we exploited. Finally in Sections VI and VII we present our opinion concerning the use of the MAC protection mechanisms and present suggestions for future experimentation.

# Previous work

The National Security Agency (NSA) along with Secure Computing Corporation (SCC) starting investigating other MAC mechanisms that would be consider the Integrity aspect of security as well as serve as a more general architecture for MAC.

The results of several previous [research projects](http://www.nsa.gov/research/selinux/background.shtml) in this area yielded a strong, flexible mandatory access control architecture called [Flask](http://www.nsa.gov/applications/links/notices.cfm?Address=http://www.cs.utah.edu/flux/fluke/html/flask.html) (See Figure 1). The architecture allows threats of tampering and bypassing of application security mechanisms to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications.

This work was not intended as a complete security solution. It was not an attempt to correct any flaws that may currently exist in an operating system. Instead, it provided an example of how mandatory access controls can be added into an operating system.

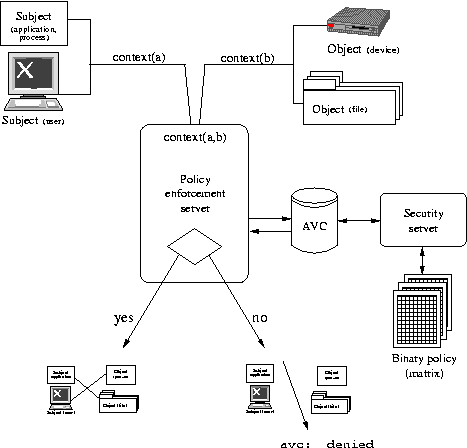


Figure 1: Flask Architecture [1]

The Flask architecture represents a flexible generic architecture that can be used to meet a wide variety of security objectives. It offers the following features:

* Processes and objects (files, sockets, etc) are assigned a security context or label. This is a set of security attributes associated with the process or object.
* Content and format of labels are dependent on the selected security model.
* Security decisions are based on these contexts. These are policy independent and have a well established interface to the object managers. This is what gives the Flask architecture its flexibility. You can plug in new policy without having to modify the rest of the kernel. Kernel must contain hooks and underlying logic to support object management.
* Clean separation of policy logic from enforcement mechanisms. Security policy logic encapsulated within a separate component of the operating system with well-defined interfaces for obtaining security policy decisions. This component is referred to as the security server.
* Enforcement components are called object managers. These are typically kernel subsystems such as process management, file system, sockets,etc). They typically receive policy information from the security server and apply policy decisions to label and control access to their objects.
* Also an access vector cache (AVC) component that stores the access decision computations provided by the security server for later use. This improves the performance of processing the security decision.
* Policy logic lies in the compiled policy matrix. Possible to implement new policy by modifying the policy and recompiling the source.
* Many different models can be used for the security server. Choice is dependent on the security objectives.

# Security Enhanced linux

An implementation of this architecture was first integrated into a security-enhanced Linux® prototype system in order to demonstrate the value of flexible mandatory access controls and how such controls could be added to an operating system. The architecture has been subsequently mainstreamed into Linux and ported to several other systems [3].

Linux was chosen as the platform for the initial implementation because of its growing success and open development environment. Linux provides an excellent opportunity to demonstrate that this functionality can be successful in a mainstream operating system and, at the same time, contribute to the security of a widely used system. A Linux platform also offers an excellent opportunity for this work to receive the widest possible review and perhaps provide the foundation for additional security research by others.

The security server portion of the Flask architecture in SELinux is implemented using a domain-type model. This type of model was chosen to demonstrate example policy since it is general enough to support many of the security objectives found in real-world security [3][4]; however, the security server is not bound to this model. A domain-type model means that every process runs in its own security domain and every resource has a type associated with it. There is a set of rules (called the security policy) which lists the actions that each domain may perform on every type. The rules are based on a combination of Identity-based access control (IBAC), role-based access control (RBAC), and Type Enforcement (TE), and optionally Multi-level security (MLS). SELinux is not however dependent on this model.

# Fedora implementation

The Fedora 11 release of Linux was chosen as the test bed for the purpose of this study. SELinux is part of the release and is enabled by default when the system is booted.

## Design

When you boot the system the first actions that init performs are mounting /proc and determining whether SELinux is enabled. It makes this determination by reading a system configuration file. This configuration file specifies the desired startup of SELinux. If it indicates that that this feature is to be disabled, then the boot proceeds as on a normal Linux system. If not a virtual file system called /selinux will be mounted, and /selinux/policyvers will be checked for the policy version that the kernel supports. The policy database /etc/selinux/X/policy/policy.YY will then be loaded into the kernel. The policy is a compiled version of the rules and contexts that will be used by the security server portion of the Flask architecture.

Also loaded at this time is a file contexts file. This file contains all the labels that will be used by SELinux to make security decisions. SELinux relies exclusively on these labels or **contexts** to make policy decisions. These security contexts contain attributes that are associated with all files and processes within in the Linux file systems. For file systems that support extended these attributes are relabeled at initialization using information stored in a global file contexts file. For files that do not support extended attributes (i.e. /proc) SELinux maintains an internal default mapping. The contexts have the following syntax in the Fedora implementation:

<user><role><type><level:category>

## Policy Options

The policy delivered with Fedora 11 includes standard tailored versions of the complete Reference Policy. The reference policy will be discussed in more detail later in this section. Currently they consist of targeted and MLS. Targeted is the default policy that is delivered with the production system.

The goal of the *targeted*policy is to lock down all processes that listen for network connections and pretty much all processes that start at boot.  These processes were viewed as the doors and windows where the hackers would enter the system. Targeted policy was born in Fedora Core 3 where about 10 domains were locked down. Other daemons on the system which do not have policy written specifically for them run in the unconfined\_t domain. Daemons and system processes running in the unconfined\_t domain only use the standard Linux Discretionary Access Control (DAC) method of access control as the SELinux policy grants all access. This is also the domain used for logged in users.

The goal of the *MLS policy* is to allow a Linux operating system to get [EAL4+/LSPP](http://www.google.com/search?q=Type%20EAL4%20LSPP%20MLS) certification. It is the first operating system to combine the [Bell and LaPadula](http://www.google.com/search?q=bell%20lapadula) model and [Type Enforcement](http://www.google.com/search?q=Type%20Enforcement%20SELinux) [8][12]. It uses all 4 fields in the file contexts to make security decisions.

The MLS policy contains rules that not only govern what security types are able to do, but also what they can do when running at a particular security level. In MLS there are two components of the Security Level - the sensitivity level and categories.

There as 16 sensitivity levels and are configured and loaded with policy using a configuration file similar to that used for file contexts.

Categories are part of an emerging policy know as Multi-Category Security. MCS policy uses the same kernel functionality and interfaces as MLS; however, this policy gives the user the empowerment to further restrict the security of their own resources. In other words, the users can label their files with categories. An example of a category is "Company Confidential". Only users with access to this category can access files labeled with the category -- assuming the existing DAC and TE rules also permit access.

## Policy editing

The Fedora 11 release did not appear to contain a lot of utilities useful in editing the standard policies. Since I did not load the ISO image for the Virtual Machine (VM) it is possible that other loadable options may exist. Instead I found a handful of command line management tools in addition to some GUI-based administration tools. I could find no policy source code or tools for creating a new binary policy; however, I did discover a GUI-based policy module tool that could create policy module files and an associated script that when executed would compile the module and dynamically load it into the kernel memory. The GUI-tool was not very intuitive and put the burden on the user to know what to enter into the text fields prior to creating the files. In my opinion, a user would have to be fairly well versed on Reference Policy before attempting such an exercise. Also, I could find no useful on-line reference material that seems to be so prevalent in many of today’s applications.

In the literature I kept finding references to a SELinux project goal that stressed an ongoing effort to make SELinux more user friendly so that the computer community would gradually gain acceptance of using the features associated with secure operating systems. If SELinux is considered to be too invasive and prevents people from doing what they want to do then they will turn it off. My experience with the GUI-based policy module tool seemed contrary to this goal.

I discovered an interesting feature called policy Booleans. This feature allows a user to enact runtime customization of SELinux policy by simply enabling a policy switch. A graphical utility as well as command line support is available to perform the customization. The main selling point for this feature is that it does not require any knowledge of the Reference Policy. The list available did not seem to be exhaustive and I believe the list is in reality a byproduct of hooks purposely compiled in with the actual loaded policy.

## Policy Customization

Tresys Technology hosts an Open Source Software site that provides a home for collaborative software development of tools and technology specific to SELinux. One of those projects is called Reference Policy. The reference policy project was created in response to the shortcomings in the original NSA example policy. The main goal was to refactor the knowledge that went into creating the original policy but apply better design principles that leverage on the strengths and features of modern SW engineering. The reference policy is the basis for nearly all SELinux policy in use today [5][8].

#### Figure 2 represents a high level overview of the layout of the Reference Policy. The policy is first divided into functional groups and within those groups are units called modules. These modules represent the smallest components of the policy. The example in Figure 2 shows the makeup of the Apache module. The Apache module is part of the services functional group.

The policy module is comprised of 3 distinct files – private, external interface, and file context. The private file is where declarations and rules local to the module are defined. These file are identified by the \*.te extension. The external interface file allows other modules access to the private types/attributes within that module. These files are identified by the \*.if extension. The file context file defines the context to use for

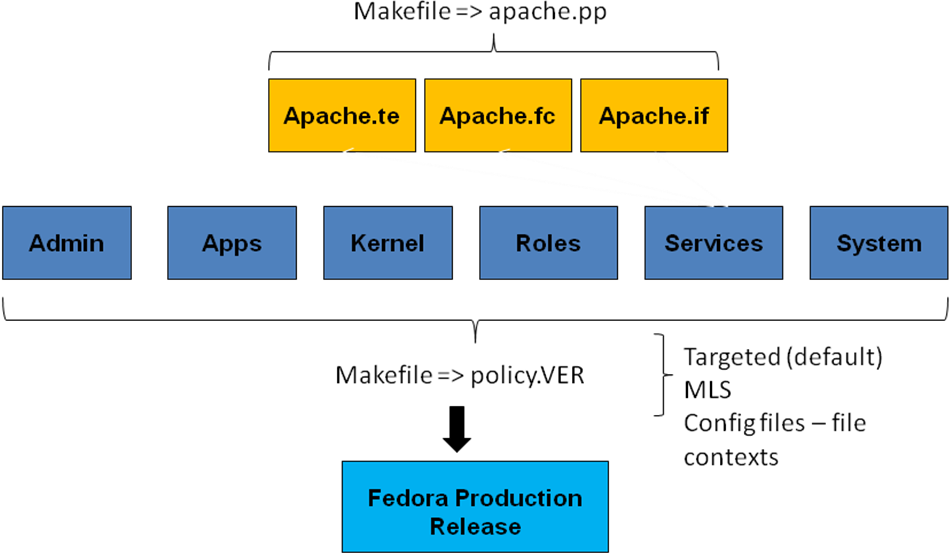


Figure 2: Reference Policy Architecture

each object. Macros are used as part of abstraction in order to simplify the policy creation process.

The policy is compiled using a Makefile. This is a multi-stage process. First the \*.*te* files are concatenated together. Next the macros are expanded using the M4 macro processor. The modules are precompiled to an intermediate file with the \*.*pp* extension. This type of file supports the loadable policy in the kernel in that a new module can be compiled and inserted dynamically during system operation. The final step of the process is to create the policy.VER binary file. This file is loaded into the /etc/selinux/X/policy directory and read by the kernel at bootup. Another file that results from the compilation is the file contexts file. This file contains the context definitions for every process and file in the Linux file systems. These file systems are initialized to these contexts at bootup time.

The size of the binary file created is dependent on the policy type. For the targeted policy this is fairly small but a strict or complete policy could equate to several megabytes of memory. This could potentially have an impact on system performance.

Creating and testing security policy is a very complicated venture. The 3 files that make up the module are developed using a special syntax that must be thoroughly understood before attempting to create or even modify a file. This process is somewhat simplified by specialized tools that available as projects on the Tresys Open Source software site. One such tool is called SLIDE. The complete reference policy and tool can be installed onto the Fedora platform using a package management tool. I will not go into detail on this tool or its installation procedure since it was part of another presentation given during the Fall 2009 CS591 class [11].

## Error Reporting

When SELinux prevents any software from accessing a particular resource, for example when Firefox is denied access to /etc/shadow, it generates a message and logs it in /var/log/audit/audit.log or /var/log/messages if audit service **is** disabled. If the log contains "avc:denied" that means it is an SELinux policy denial. Note that you would need administrator privileges (root access) on your system to be able to read this log file. There are command line and GUI-based tools that can aid in the interpretation of these messages.

# Fedora analysis

Figure 3 depicts the content of the main configuration file that SELinux references at bootup. It controls how SELinux operates and on which policy it loads into the kernel memory.

The policy binary contained in the targeted directory will be loaded. The configuration file also specifies that SELinux will be enabled and in enforcing mode. Enforcing means that all the processes targeted for security protection will be passed through the policy logic and accesses decisions made based on those policies. Any denials will be written to the system logs. There is also another mode of operation called permissive mode. This mode differs from enforcing in that it only logs denials. The literature recommends this mode of operation when first checking out a new policy.

Figure 4 depicts a typical context as applied to users. The operating system recognizes a Linux user but when SELinux is enabled that user is mapped to an internal SELinux user. The SELinux user is the identity known to the policy that is authorized for a specific set of roles. Linux users are defaulted to unconfined\_u.

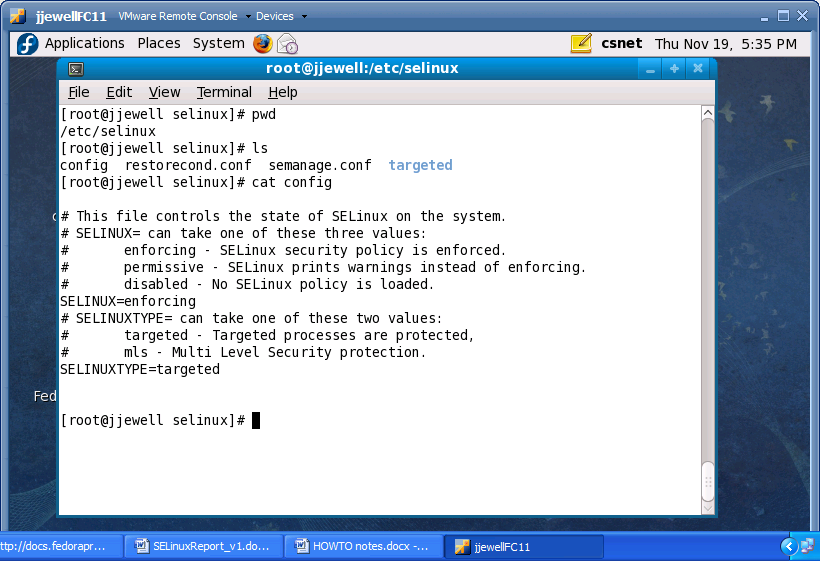


Figure 3: SELinux configuration settings

An interesting confinement of the user is illustrated when a logged in user tries to elevate permission by logging in as the super user In the DAC system, that user would have dangerous power to intentionally or unintentionally comprise the system security. Under SELinux, the user’s context does not change when the super user account is invoked. This is because SELinux relies primarily on the role attribute associated with users. The role can only be changed in certain ways. One way is to simply login as an account that has the desired role. The other way is to use the “newrole” command to assume a new role and hence a new context. Note that this command will allow the user to change roles to those that have been mapped to the default identity.

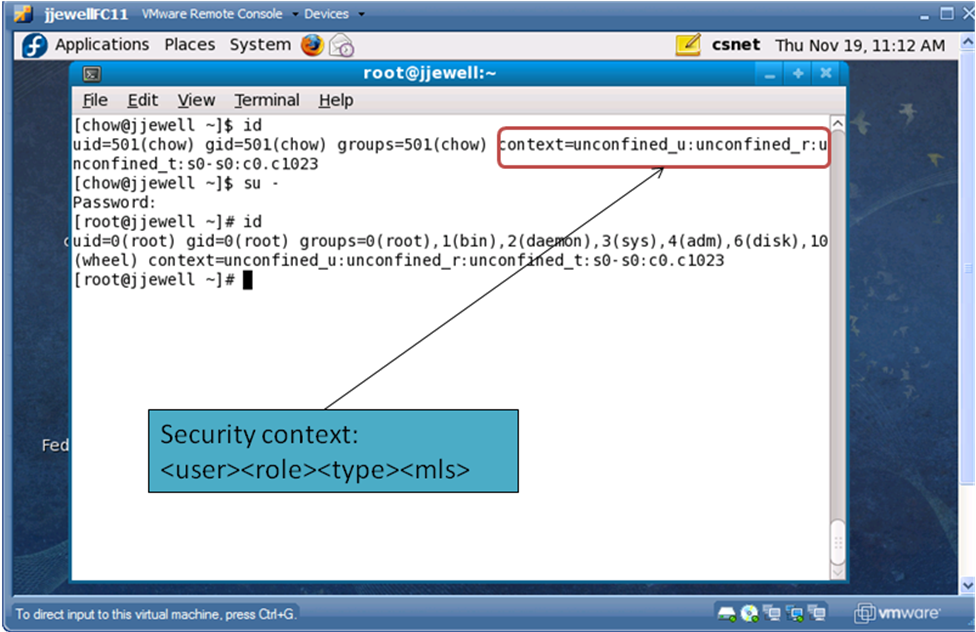


Figure 4 - Illustration of user context

Figure 5 illustrates the role and type attributes. A role determines what domains a user can access. There must be a rule that allows an unconfined user to access a confined domain. For example a rule in the password module might specify the following:

*role unconfined\_r types passwd\_t*

This rule allows the user access to the password domain and can execute programs associated with that domain. Figure 5 shows the user running the passwd command in the background. A “ps” with the “-Z” option will show the context associated with the passwd process. Note that the process is of type “passwd\_t”. The fact that the process is running is the proof that the above rule was honored. If not an AVC message would have been generated a logged in the system log.

Every process runs in a domain. A domain is basically a list of what processes can do, or what actions a process can perform on different types. Think of a domain like a standard Unix uid. Say root has a program and does a *chmod 4777* on that program (making it setuid root). Anyone on the system,

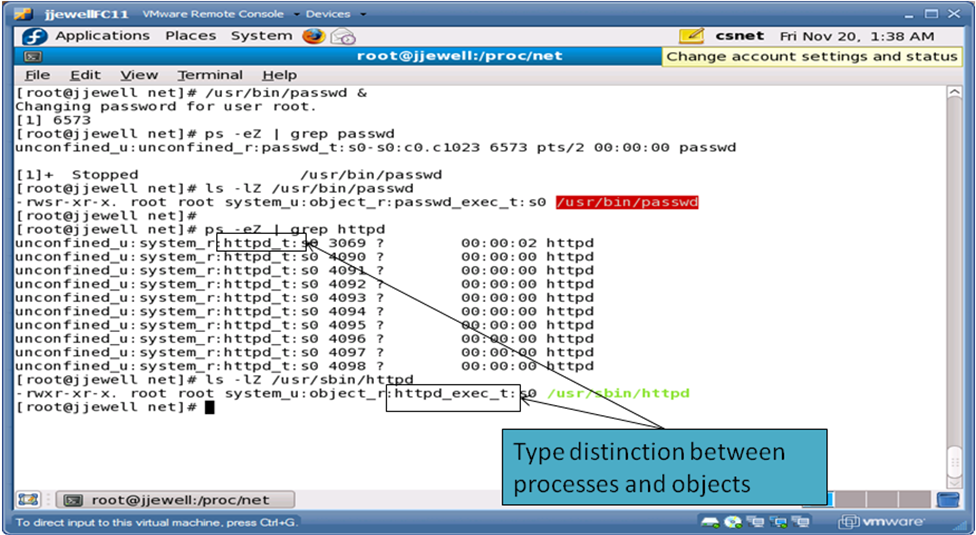


Figure 5: Illustration of role and type attributes

even the nobody user, can run this program as root thereby creating a security issue. With SE Linux however, if you have a process which triggers a domain transition to a privileged domain, if the role of the process is not authorized to enter a particular domain, then the program can't be run. These concepts are what give SELinux its power but also make it complicated from a user perspective.

A type is assigned to an object and determines who gets to access that object. The definition for domain is roughly the same, except a domain applies to process and a type applies to objects such as directories, files, sockets, etc. But if you look in /proc where most of the process files lives and do an ls –Z you will find that it also has a type. Even though proc represents running processes, the entries under /proc are labeled as types. IMPORTANT CLARIFICATION – the type is also the domain that the process runs in.

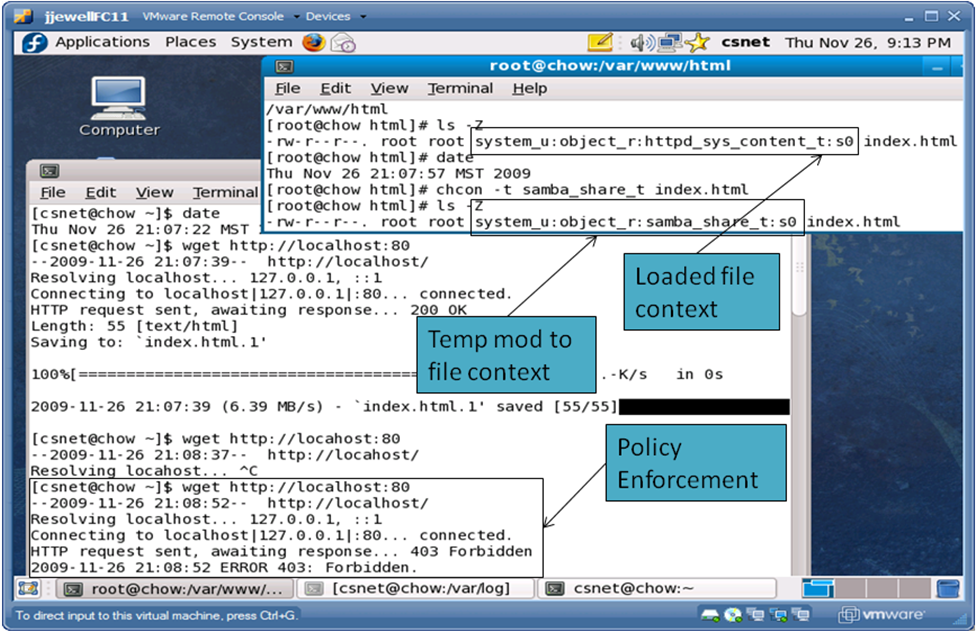


Figure 6: Illustration of type enforcement

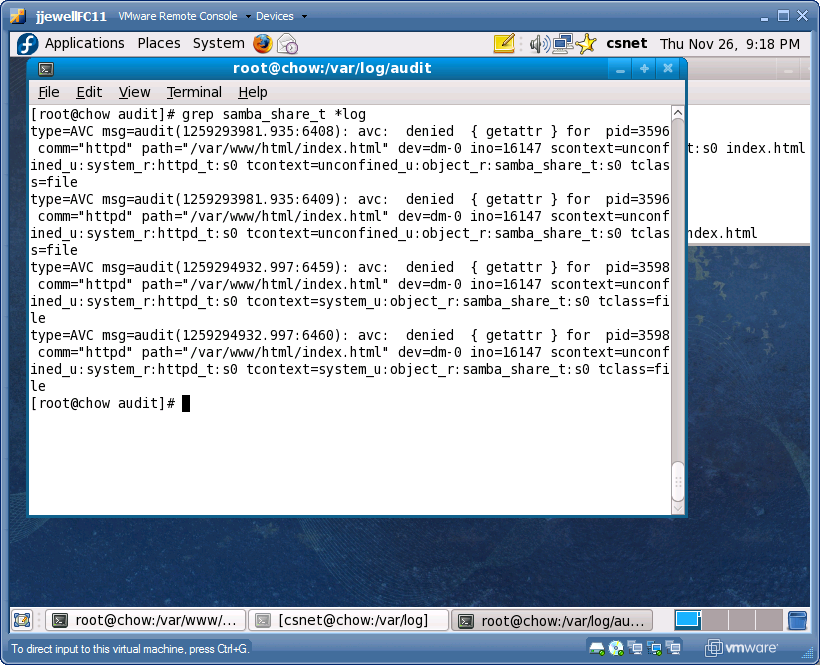


Figure 7: Illustration of AVC denial message

Some examples of domains are sysadm\_t which is the system administration domain, and user\_t which is the general unprivileged user domain. *init* runs in the init\_t domain, and *named* runs in the named\_t domain.

Figures 6 and 7 represent an example of type enforcement. The scenario is to read a web page using the text-based wget tool. The index.html file shows a type of httpd\_sys\_content\_t. A rule currently exists in the policy that allows the processes running in the httpd domain (type httpd\_t) to read files of the above type. In this case the wget command is successful. We next change the type of the index.html file to something for which a rule does not exist. When we try to execute the wget command we get a message stating “Error – FORBIDDEN”. Figure 7 shows the corresponding AVC denial message.

# Conclusion

The Fedora 11 implementation containing SELinux certainly peaks the users interest in the level of fine grained control of any process or object but is still not as user friendly in my opinion as the developers would probably like. The heart of the MAC mechanism lies in the Reference Policy but without a lot of training and familiarization it seems that a user of the application cannot fully exploit the powerful features available.

The out-of-the-box canned policies are not easily modified given the delivered toolset. Improvements in on-line help with example documentation would go along way in helping users get up to speed without searching all over the web for information.

# Future Work

* Testing exploits with and without SELinux. The following link

<http://magazine.redhat.com/2008/02/26/risk-report-three-years-of-red-hat-enterprise-linux-4>

describes some of the exploits encountered over several years of SELinux operation and how the security rules helped to easily mitigate these threats.

* Install and experiment with MLS policy. Setup various user accounts and demonstrate that certain users can only access certain classifications of information given their clearance.
* Compare user experience on other platforms such as Debian, Ubuntu, etc.

##### References

1. Red Hat documentation (2009). SELinux Architectural Overview. Retrieved Nov. 5, 2009 from <http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/selinux-guide/selg-chapter-0013.html#SELG-SECT1-0014>
2. P. Loscocco, Smalley S. Muckelbauer, P, Taylor, R, Turner S., and Farrell, J. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environment. Proceedings of the [21st National Information Systems Security Conference](http://csrc.nist.gov/nissc/1998/proceedings), pages 303-314, October 1998.
3. P. Loscocco and Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System. Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01), June 2001
4. P. Loscocco and Smalley S. Meeting Critical Security Objectives with Security-Enhanced Linux. Proceedings of the 2001 Ottawa Linux Symposium
5. J. PeBenito, F. Mayer, and K. MacMillan Reference Policy for Security Enhanced Linux. 2006 **Security** **Enhanced** **Linux** Symposium
6. [Wikipedia (2009). Manadatory Acce Control Retrieved Nov 10, 2009 from http://en.wikipedia.org/wiki/Mandatory\_Access\_Control](http://en.wikipedia.org/wiki/Mandatory_Access_Control)
7. [Widipedia (2009). Discretionary Access Control. Retrieved Nov 10, 2009 from http://en.wikipedia.org/wiki/Discretionary\_access\_contro](http://en.wikipedia.org/wiki/Discretionary_access_contro)l
8. [Tresys Open Source Software (2009).](http://oss.tresys.com/projects/refpolicy) SELinux Reference Policy. Retrieved Nov 10, 2009 from <http://oss.tresys.com/projects/refpolicy>
9. (2009) . Writing SE Linux policy HOWTO. Retrieved Nov 15, 2009 from <http://www.lurking-grue.org/writingselinuxpolicyHOWTO.html>
10. Fedora Project (2009). Fedora 11 SELinux user’s guide. Retrieved Nov 15, 2009 from http://docs.fedoraproject.org/selinux-user-guide/f11/pdf/Security-Enhanced\_Linux.pdf
11. *UCCS CS591 (2009) Retrieved Dec 10, 2009 from* <http://cs.uccs.edu/~cs591/studentproj/projF2009/sjahnke/doc/>

[SELinuxSLIDEsjahnke.pptx](http://cs.uccs.edu/~cs591/studentproj/projF2009/sjahnke/doc/SELinuxSLIDEsjahnke.pptx)

1. [C. Hanson. SELinux and MLS: Putting the pieces together.](http://selinux-symposium.org/2006/papers/03-SELinux-and-MLS.pdf) Proceeding from the 2006 SELinux Symposium