An Introduction to the NSA's Security-Enhanced Linux: SELinux

Susan Rajnic

An Introduction to the NSA's Security-Enhanced Linux: SELinux
By Susan Rajnic

Abstract

This paper will introduce the NSA's research project termed "Security-enhanced" Linux. It has been recognized that securing applications is only half of the battle: a computer system must also employ security policies at the OS level, and the current model of user vs. administrator that we find in standard Unix is insufficient. Security-enhanced Linux, or "SELinux", is defined as "enforc[ing] mandatory access control policies that confine user programs and system servers to the minimum amount of privilege they require to do their jobs"(1). SELinux is neither a tool for encryption nor a full distribution of Linux; instead, it is a modification of the kernel to include a "security server". This internal security server is responsible for implementing a configurable security policy to the way processes and users are allocated system resources and permissions. SELinux derives its architecture from a previous project called the "Flask" operating system. This paper will assume that the reader possesses working knowledge of the Unix operating system, and understands the role of Linux in the Unix world.

Introduction

The NSA, or National Security Agency, is a US federal agency that exists to compromise the confidentiality of information of foreign adversaries, while protecting that of the United States. It is therefore easy to see their interest in computer-based security and their involvement in developing improved system security mechanisms. Under the Information Assurance Directorate, or IAD, the NSA conducts several research projects aimed at filling in the security "gaps" of commercially available software and hardware(2). SELinux is one of these projects, and its homepage can be found at http://www.nsa.gov/selinux/index.html.

The main goal of this project is to address the problem of information separation at the operating system level. Research documentation tell us that SELinux "provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements"; it continues to explain that "[t]his 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"(3). Documentation repeatedly tells us that it does this by incorporating a "strong, flexible mandatory access control architecture"(3). But what exactly does this mean?

What is mandatory access control?

To answer this question, one must understand the type of access control available in most common distributions of Linux. Unix distributions without security enhancement have two types of users: the standard user and the super user, also known as the root user. When a standard user executes a program, the program will usually run within the permission boundary of that user. However, some programs need more permissions to do their job than the executing user is allowed. In this case, the program is considered

a “setuid” program. Setuid programs are programs which when run by a normal user, change their user id to 0 (root) and then perform tasks at that level. The passwd command is typically given as an example of a setuid program. The user wishing to change their password should not have access to edit files associated with password controls; therefore, although the passwd command is executed by a common user, it must receive the privileges of the super user in order to complete its task. This is what is referred to as “coarse-grained” privileges. The setuid program, when accepting root access and therefore full reign of the OS, is most likely accepting far more privileges than it actually needs. Just because the passwd command needs to edit the /etc/passwd or /etc/shadow file, it does not need the ability to add or remove cron jobs, for example. If we could restrict the setuid commands to only accept the greater privileges that they absolutely need in order to perform their specific task, then we eliminate or reduce the harm of a flawed or malicious setuid program. This is what is referred to as “fine-grained” privileges.

The flaws in the Unix security model are not confined to setuid programs. Even when a process does not need permissions beyond the executing user, the fact that the process has access to everything attributed to that user is dangerous enough. As stated by the main contributors to the SELinux project, “[a]s long as users have complete discretion over objects, it will not be possible to control data flows or enforce a system-wide security policy”(3). This current privilege-granting scheme based solely on user ids and user ownership is known as Discretionary Access Controls (DAC). An answer to its problems is a model based on Mandatory Access Controls (MAC). A MAC architecture grants limited or “fine-grained” privileges to users and processes and “is considered to be any security policy where the definition of the policy logic and the assignment of security attributes is tightly controlled by a system security policy administrator”(4). This means that the system administrator will be able to determine what access is granted to executable code, regardless of the privilege set granted to the executing user. Complexity is introduced, but improvements to security are gained.

The Flask Architecture

The way in which flexible, strong, mandatory access controls are implemented in SELinux is based on what they call the Flask architecture. Flask stands for “Flux Advanced Security Kernel” but defining Flux and its predecessors is beyond the scope of this paper. All one needs to know in order to grasp SELinux is that it incorporates the work of the NSA, the Secure Computing Corporation, and the University of Utah, who share a common interest in MAC vs. DAC operating systems. Flask at one point was its own microkernel-based operating system, but now its security architecture has been ported to a Linux base, in the form of SELinux, to introduce it to the open source community and gain a wider audience. Another benefit of migrating to Linux is the demonstration that Flask architecture could work in a “real live” OS as opposed to microkernel-based systems whose limitations could be used as an argument against the strength of Flask.

Flask is not the only operating system that is based on the concept of MAC. What

makes its architecture special is the concept of a “flexible” security policy. If implementing fine-grained access rights introduces a complex map of privileges and roles, then it is not prudent to assume that all users of SELinux will want to follow the same map, or that the needs of the system will never change while in use. The security policy should be designed by the system administrator of the system to suit the needs of his organization and users. The secure OS should provide the tools to implement this policy, regardless of its content. In order to accommodate flexibility, the security policy logic and the apparatus which implements this logic must be separated. The following diagram and caption are from the University of Utah’s documentation of Flask and serve to illustrate this separation(5):

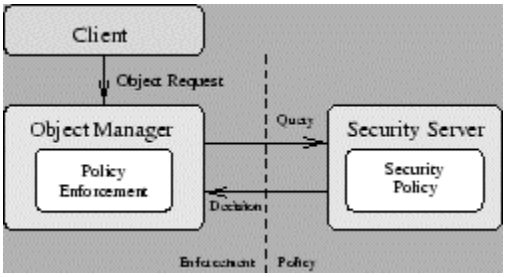

Figure 1: The Flask architecture. Components which enforce security policy decisions are referred to as *object managers*. Components which provide security decisions to the object managers are referred to as *security servers*. The decision making subsystem may include other components such as administrative interfaces and policy databases, but the interfaces among these components are policy-dependent and are therefore not addressed by the architecture.

For system administrators who are accustomed to always having a configuration file to tweak, this may not seem such a radical idea. However, implementing true *operating system level* security that can serve any policy is extremely difficult, as we shall see. Immediately three design necessities are introduced: a mechanism to control the distribution of security rights; a mechanism to enforce the fine-grained access; and a mechanism to revoke previously granted access rights.

The first design requirement is satisfied because the security server analyses the security policy for each and every decision it is requested to make. Nothing gets done without the security server’s permission. The latter two design requirements are addressed within the object manager, as seen in figure 1. More than one object manager exists within the system, because the object manager contains labeling and “handling” routines specific to the types of objects within that manager. Managing the security policy and possible policy changes of a file or a directory is very different than managing the security policy of network sockets, for example.

In order for an object to obey its policy, it needs to know its policy. This is referred to as the object’s “security context” and exists as a policy-independent data type in the form of a variable length string. The system administrator has the ability to define any number of security contexts, as long as they can be stored as variable length strings. Where these security contexts are defined will be explained in the section covering installation. However, using a variable-length string for lookups of an object is inefficient. Therefore, an object also has a “security identifier”, or SID, which is another policy-independent data type in the form of a 32-bit integer. The security server randomly creates these SIDs, so there is no creation algorithm to be spoofed. Also, the SIDs are not fixed and will change if the server is rebooted. A SIDs is assigned in the following manner: a client (possibly a user on the system) wants to create an object

(create a file, for example). The command used to create this file knows to ask file system object manager for the new file object. The object manager knows it must ask the security server for the security context and SID, and does this by sending the SID of the user to the security server. For, the security server must know who is requesting the new file, because the requesting client's security context will affect the security context of the object being created. The security server looks at its policy, determines a security context, and returns an appropriate SID for the new object.

Does this sound inefficient to you? For each and every action on the server, an object manager is making a request to the security server, and the security server is crunching away to obtain a security decision. Flask addresses this by implementing a caching system called "access vector cache", or AVC, available to all object managers. So in reality, when our user wants to create a file, the file system object manger first checks the AVC to see if there are cached permission values for this particular user and the file object. If nothing is cached, the security server will take the user's SID, the request to create a file, and will return all possible file permissions in the form of an access vector. Now, when the the same user wants to delete this file, instead of asking the security server for permission, the object manager can consult it's "cheat sheet" in the AVC. When the security server gave permission to this user to create a file, it also returned the information that, under the same conditions, this user may or may not delete certain files, for example. Obviously the AVC is a boost to performance, but a hindrance to dynamic changes in security policy.

So what happens when the security policy changes? What happens when the system administrator decides that our file-creating user is no longer allowed to create files? Basically, a protocol exists between the object managers and the security server whereby the security server alerts the object managers of changes, and the object managers alert the security server when they've completed these changes. Within this protocol are requirements that object managers perform these steps very quickly, eliminating the possibility of dangerous code preventing the object manager of making policy changes through repeated object request calls, for example. While the synchronizing protocol is performing these steps, the AVC is updating its cache. It then plays a role in revoking "migrated" permissions. Revocation of migrated permissions involves a complicated management of thread state and kernel state. Detailed explanations can be found in (5).

Flask in Action

It all sounds a bit academic without seeing it in action. Therefore, I will now detail the basic installation instructions for SELinux so that you can get it up and running to test for yourself. Note that SELinux has been developed exclusively with the RedHat distribution and currently supports up to version 7.2. If you prefer another Linux distribution, further customization may be required than is documented. Other distributions will most likely work, but have not been officially tested by the NSA.

Begin by downloading the source code files from

http://www.nsa.gov/selinux/download.html. You have the option of selecting the stable 2.4 version, or the 2.5 prototype. If you follow the SELinux download link to the version 2.4 download page, you are given five different download options:
(a) "everything" which is a 2.4.17 kernel including the LSM (Linux Security Module) patch, and the extra files and utilities needed to run SELinux;
(b) 2 separate downloads where one contains the 2.4.17 kernel with LSM patch and the other is the files needed by SELinux;
(c) option (b) minus the kernel, or just the patch and the SELinux files
(d) patches and new code only
(e) all the patches and new code together

If you're new to Linux, new to security enhanced operating systems, or have simply never recompiled a kernel before, then this page may not make immediate sense. You'll need to have a basic understanding of two things: loadable kernel modules, and the LSM, or Linux security module.

A loadable kernel module is just a piece of the kernel that you may, or may not, need. If you need this piece, you can load it into your kernel. If you don't, you can leave it out. LKM's are compiled separately from the kernel and can be loaded or linked into a running kernel. A good example of a loadable kernel module is a device driver for PCMCIA. Unless you're running Linux on a laptop, why include this driver? However, if you're using your desktop machine to build a custom kernel for your laptop, you have the option of linking in PCMCIA support. The LSM is simply a LKM to support security enhancements to Linux. It is not built to support only SELinux. In fact, the creation of the LSM owes much of its existence to the work of Dr. Crispian Cowan and his involvement with another security-enhanced Linux distribution called "Immunix", http://www.immunix.org/. On the LSM mailing list, Dr. Cowan posts comments made by Linus Torvalds which inspired production of the LSM, and relate the desire to keep any Linux security enhancement module non-implementation specific. Meaning, Mr. Torvalds does not care if we, the system administrators, prefer SELinux or Immunix – he is only interested in including a security module that supports choice and offers flexibility. This post can be found on the mailing list archive here: http://mail.wirex.com/pipermail/linux-security-module/2001-April/000005.html.

It is worth nothing that the LSM offers support for security enhancements mainly by implementing "hooks", which offer an interface to catch and audit system calls and functions.(6) It is then up to SELinux or Immunix or any other security enhancement to use these hooks from within their security architecture. It is also worth noting that the LSM is not truly a loadable module, yet. This is why it is refered to as a "patch" on the download page. Dr. Cowan addresses this in a LSM mailing list post that can be found at this link: http://www.infodrom.org/Mail-Archive/security/2001/0014.html

Now that we know what LSM is and how SELinux may work together with it, it's time to get our hands dirty. For this example, I opted to download the full 36Mb compressed tar file offered as the first download option. This is the best option for those new to kernel customizations. I chose to download the file to /root, as I was logged into console as

the root user. If you want to place it somewhere else, please adjust the following commands accordingly. Perform the following to decompress and separate:

% gunzip lsm-2.4-selinux-2002011718.tgz
% tar –xvf lsm-2.4-selinux-2002011718.tar

This has now created two directories: /root/lsm and /root/selinux. Following the superb instructions found in /root/selinux/README(7), I have performed the "QUICK INTSALL". This README document also includes lengthy "building" and "installing" sections for those who wish to further delve into the process, but to get up and running we'll begin with this quick solution, save a few minor changes. The most notable variation on these README pages is that I did not remove the test users from the configuration files as recommended. The configuration files specify two users on the system: jadmin, a user with root privileges, and jdoe, a user without root privileges. Instead of trying to map existing users to several configuration files, I simply created jadmin and jdoe accounts for the sake of simplicity. If you choose to do this as well, please remember to add jadmin to the group "root" in /etc/group.

1. If you're running a GUI such as GNOME or KDE, and you boot into a graphical login, you'll have to change this and prepare to boot into a non-graphical mode. You can do this by editing your /etc/inittab file and changing the default runlevel from 5 to 3. The first line should look something like this: id:3:initdefault: If you want to run your X server with SELinux, see step 5 in the selinux/README file for more information.

2. As mentioned, I performed this installation on a RedHat 7.2 workstation. If you are using RedHat 7.2 as well, you will need to edit the selinux/utils/Makefile, and change the LOGROTATE_VER variable as follows:
   #LOGROTATE_VER=3.5.4-1
   LOGROTATE_VER=3.5.9

3. Make sure that you are root or have root privileges, and type the following command at the command prompt within the selinux directory:
   % make quickinstall

4. You'll now be presented with an interface for configuring the new security-enhanced kernel. If in doubt, leave the default choices. I've found that I only had to add support for the ext3 file systems, found under the "File systems→" menu choice. However, this is because I had formatted my Linux partitions as ext3 partitions. If you did not do this, then you will not need to enable ext3 support. Not sure if you have ext3 file systems or not? Use the command
   % mount
   to get a list of mounted file systems and their file system type.

5. After you exit the kernel configuration menu and save your changes, the quickinstall will run and compile for quite some time. Upon completion, you will find a new image in your /boot directory called "vmlinuz-2.4.17-selinux". Now we'll need to edit

the boot loader to add this kernel as a bootable option. If you are using the LILO boot loader, simply add the following lines to your /etc/lilo.conf file:

```
Image=vmlinuz-2.4.17-selinux1
Label=SELinux
```

Do not add an initrd option to this entry, and do not overwrite or delete your original linux kernel entry. If something should go wrong with this boot, you will have a non-security-enhanced kernel to boot back into. After editing /etc/lilo.conf, remember to run /sbin/lilo at the command prompt to update the boot loader. If you are using GRUB as opposed to LILO, please see step 12 in selinux/README documentation.

6. Now we're ready to boot into our security enhanced kernel. Reboot, select "SELinux" at the LILO prompt and notice the following message upon login:

   ```
   Your default security context is root:sysadmin_r:sysadmin_t
   Do you want to enter a new security context?[n]
   ```

   This message will time out and return to the login prompt if ignored. Because you're not yet familiar with the out-of-the-box security policy, press enter to accept the default security context. The README file notes that you will only get the option to change your security context if you are logging in at console. If you are logging in remotely via sshd, you will be forced to login under the default context and change this context with the program "newrole".

7. As root and within the selinux directory, perform the following commands to assign proper security labels to files created by the non-secure version of Linux as it was shutdown:

   ```
   % cd setfiles
   % make verbose
   % cd ..
   ```

8. Finally, we'll need to add the /usr/local/selinux/bin directory to our path. In this directory are modified versions of several different system commands such as find, ls, and cp, as well as some Flask specific commands such as chsid and chcon. I like to add this directory to the beginning of my $PATH variable with the following command:

   ```
   % export PATH=/usr/local/selinux/bin:$PATH
   ```

   The modified versions of common commands include some command line options specific to SELinux, such as being able to show the security context or SID for processes, files, users, etc.

Now that SELinux is up and running, let's examine the difference between this OS and

plain old RedHat. The following examples will only serve to peek into the world of SELinux policy, and uncover the tip of the security policy iceberg. A complex web of permissions exists between processes, files, and users, and a full explanation of how this is implemented is explained in (8).

We'll begin by performing a simple ps command. If we were in non-secure Linux, a typical ps –e command would look like this:

```
PID TTY          TIME CMD
    1 ?        00:00:04 init
    2 ?        00:00:00 keventd
    3 ?        00:00:00 kapm-idled
    4 ?        00:00:00 ksoftirqd_CPU0
    5 ?        00:00:00 kswapd
    6 ?        00:00:00 kreclaimd
    7 ?        00:00:00 bdflush
    8 ?        00:00:00 kupdated
    9 ?        00:00:00 mdrecoveryd
   13 ?        00:00:00 kjournald
   88 ?        00:00:00 khubd
  181 ?        00:00:00 kjournald
  721 ?        00:00:00 syslogd
  726 ?        00:00:00 klogd
  746 ?        00:00:00 portmap
  774 ?        00:00:00 rpc.statd
  887 ?        00:00:00 apmd
  956 ?        00:00:00 xinetd
  996 ?        00:00:00 sendmail
 1015 ?        00:00:00 gpm
 1033 ?        00:00:00 crond
 1085 ?        00:00:00 xfs
 1121 ?        00:00:00 atd
 1135 tty1     00:00:00 login
 1136 tty2     00:00:00 mingetty
 1137 tty3     00:00:00 mingetty
 1138 tty4     00:00:00 mingetty
 1139 tty5     00:00:00 mingetty
 1140 tty6     00:00:00 mingetty
 1143 tty1     00:00:00 bash
 1191 tty1     00:00:00 ps
```

In SELinux, when we execute /usr/local/selinux/bin/ps –e --context, we have an option of seeing the security context for each process and our output is now this:

```
PID    SID CONTEXT                              COMMAND
    1      7 system_u:system_r:init_t             init [3]
    2      7 system_u:system_r:init_t             [keventd]
    3      1 system_u:system_r:kernel_t           [ksoftirqd_CPU0]
    4      1 system_u:system_r:kernel_t           [kswapd]
    5      1 system_u:system_r:kernel_t           [bdflush]
    6      1 system_u:system_r:kernel_t           [kupdated]
    7      7 system_u:system_r:init_t             [khubd]
    8      7 system_u:system_r:init_t             [kjournald]
  130    187 system_u:system_r:mount_t            [kjournald]
```

```
442    188 system_u:system_r:syslogd_t              syslogd -m 0
447    198 system_u:system_r:klogd_t                klogd -2
467    199 system_u:system_r:portmap_t              portmap
495    200 system_u:system_r:rpcd_t                 rpc.statd
659    204 system_u:system_r:inetd_t                xinetd -stayalive -reuse -
699    205 system_u:system_r:sendmail_t             sendmail: accepting connec
718    207 system_u:system_r:gpm_t                  gpm -t ps/2 -m /dev/mouse
736    208 system_u:system_r:crond_t                crond
788    210 system_u:system_r:xfs_t                  xfs -droppriv -daemon
824    211 system_u:system_r:atd_t                  /usr/sbin/atd
836    213 system_u:system_r:local_login_t          login -- root
837    212 system_u:system_r:getty_t                /sbin/mingetty tty2
838    212 system_u:system_r:getty_t                /sbin/mingetty tty3
839    212 system_u:system_r:getty_t                /sbin/mingetty tty4
840    212 system_u:system_r:getty_t                /sbin/mingetty tty5
841    212 system_u:system_r:getty_t                /sbin/mingetty tty6
844    214 root:sysadm_r:sysadm_t                   -bash
910    214 root:sysadm_r:sysadm_t                   ps -e --context
```

A security context has the syntax user:role:domain or user:role:type. This is the variable length string mentioned earlier in the explanation of Flask. Following this syntax, it seems that my ps command is running as user “root”, role “sysadm_r”, and domain “sysadm_t”, because I am executing this as the root user. However, the crond daemon, for example, is running not as root, but as user “system_u”. The file selinux/policy/users, where we define security contexts for our users, tells us that system_u “is the user identity for system processes and objects”. It continues: “there should be no corresponding Unix user identity for system_u, and a user process should never be assigned the system_u user identity”. We are now beginning to see the concept of “fine-grained” access in practice. All system processes are running not as root, but as system_u, within a system role, within their own domain. The process crond is working in domain “crond_t”, inetd is running within domain “inetd_t” and so on.

Security contexts are defined across a multitude of configuration files found in selinux/policy/domains. Within this directory is a directory for each “admin”, “program”, “system”, and “user”. These directories contain files with a “.te” extension, standing for “type enforcement”, which defines the declarations and rules for each domain. There is also a file in the domains directory called “every.te” with rules applying to every domain. Domains are subdivided into types, and can be understood by example. A good example from (8) explains that the syslogd daemon, as we can see above, is running under the syslogd_t domain. However, if we use the SELinux version of “ls” in the /sbin directory with the command “ls --context”, we can see that the syslogd executable has the security context of system_u:object_r:syslogd_exec_t. Furthermore, the file that this daemon writes to, /dev/log, has a security context of system_u:object_r:devlog_t. Each functioning piece of this process is given attributes and permissions subject to it’s minimal amount of needs. These attributes are explicitly stated in the file selinux/policy/domains/system/syslogd.te .

So we’re beginning to see how SELinux may protect us from malicious daemons and setuid programs. What about when a user executes malicious code? How safe are the

resources available to that user? The permission sets of domains and types are also defined for users. When are test user logs in, “jdoe”, her security context is jdoe:user_r:user_t. However, if jdoe fires up a GUI and begins to surf the web with the mozilla browser, is that browser executing as jdoe? A ps –e --context shows us that jdoe’s web browser is executing as jdoe:user_r:user_netscape_t. Therefore, this executable does not have access to everything owned by jdoe. It’s permissions are mapped in a policy configuration file in selinux/policy/domains/user/user.te.

Further explanations of Flask architecture in SELinux require knowledge of object-oriented programming and advanced knowledge of the Linux OS. This paper serves only to introduce you to the main concepts of fine-grained access and configurable policy control at the OS level. The NSA website contains further technical specifications of SELinux, as well as man pages for each program in /usr/local/selinux/bin.

Conclusions

SELinux is not recommended for any level of implementation other than development and testing. It is not approved for government use. It is a work-in-progress of a new concept for OS design. SELinux does not attempt to address all known security issues, but it does offer a framework for preventing Unix processes from unauthorized behavior such as reading other process data, changing data, bypassing coarse-grained security, or interfering with other processes. By examining the model of user:role:domain, and how domains can be further subdivded into types, we can imagine how malicious or flawed programs are prevented from doing damage on a system-wide or account-wide basis. In reviewing the Flask architecture and the separation of the security server and the object control, we can see how a system can be configured to support various patterns of domains and types. Obviously, the task is monumental, but now that the project is being conducted under the terms of the GNU General Public License, and the Linux Security Module is being developed, we can expect much activity in this area in the coming years.

Quoted Sources:

1. Security-Enhanced Linux Frequently Asked Questions (FAQ)
http://www.nsa.gov/selinux/faq.html

2. About the IAD, Delivering IA Solutions for Cyber Systems
http://www.nsa.gov/isso/brochure/index.htm

3. Loscocco, Peter A.; Smalley, Stephen D. *Meeting Critical Security Objectives with Security-Enhanced Linux*
http://www.nsa.gov/selinux/ottawa01-abs.html

4. Loscocco, Peter A.; Smalley, Stephen D.; Muckelbauer, Patrick A.; Taylor, Ruth C.; Turner, S. Jeff; Farrell, John F.; *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments*
http://www.cs.utah.edu/flux/fluke/html/inevit-abs.html

5. Spencer, Ray; Loscocco, Peter A.; Smalley, Stephen; Hibler, Mike; Andersen, David; Lepreau, Jay; *The Flask Security Architecture: System Support for Diverse Security Policies*
http://www.cs.utah.edu/flux/papers/flask-usenixsec99-abs.html

6. PRFW: hook system; http://www.freesoftware.fsf.org/jailuser/

7. QUICK INSTALL README, also found at
http://www.nsa.gov/selinux/doc/readme.html

8. Loscocco, Peter A.; Smalley, Stephen D. *Integrating Flexible Support for Security Policies into the Linux Operating System*
http://www.nsa.gov/selinux/freenix01-abs.html

Consulted Sources:

IBM Linux Technology Center : Kernel Hooks (GKHI) for Linux:
http://oss.software.ibm.com/developer/opensource/linux/projects/gkhi/HOWTO

Linux-security-module mailing list archive: http://mail.wirex.com/pipermail/linux-security-module/2001-April/000005.html

Infodrom Oldenburg Security: Linux Security Module Interface:
http://www.infodrom.org/Mail-Archive/security/2001/0014.html

Smalley, Stephen; Fraser, Timothy; *A Security Policy Configuration for the Security-Enhanced Linux*
http://www.nsa.gov/selinux/doc/policy.pdf

DiBona, Chris. “Security Enahnced (SE) Linux, The OS that Came in from the Cold”; Linux Magazine, September 2001
http://www.linux-mag.com/2001-09/se_linux_01.html

Weiss, Todd R. “Feds unveil 'security-enhanced' Linux prototype”; Computerworld, January 2001
http://www.computerworld.com/cwi/story/0,1199,NAV47_STO56110,00.html

NAI Labs Advanced Research
http://opensource.nailabs.com/selinux/index.html

Slashdot discussion boards
http://slashdot.org/articles/00/12/22/0157229.shtml

List of other security enhanced operating systems
http://www.wiretapped.net/indexes/operating-systems.html

Flask: Flux Advanced Security Kernel
http://www.cs.utah.edu/flux/fluke/html/flask.html

Last Updated: September 29th, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS October Singapore 2020	**Singapore, SG**	**Oct 12, 2020 - Oct 24, 2020**	**Live Event**
SANS Community CTF	**,**	**Oct 15, 2020 - Oct 16, 2020**	**Self Paced**
SANS Tel Aviv November 2020	**Tel Aviv, IL**	**Nov 01, 2020 - Nov 06, 2020**	**Live Event**
SANS Sydney 2020	**Sydney, AU**	**Nov 02, 2020 - Nov 14, 2020**	**Live Event**
SANS Secure Thailand	**Bangkok, TH**	**Nov 09, 2020 - Nov 14, 2020**	**Live Event**
APAC ICS Summit & Training 2020	**Singapore, SG**	**Nov 13, 2020 - Nov 21, 2020**	**Live Event**
SANS FOR508 Rome 2020 (in Italian)	**Rome, IT**	**Nov 16, 2020 - Nov 21, 2020**	**Live Event**
SANS Community CTF	**,**	**Nov 19, 2020 - Nov 20, 2020**	**Self Paced**
SANS Local: Oslo November 2020	**Oslo, NO**	**Nov 23, 2020 - Nov 28, 2020**	**Live Event**
SANS Wellington 2020	**Wellington, NZ**	**Nov 30, 2020 - Dec 12, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**