University of Piraeus

Department of Digital Systems

Systems Security Laboratory

M.Sc. in “Techno-economic Management & Security of Digital Systems”

Master Thesis

Development of a Secure Linux Distribution

Panagiotis Tsesmetzis

February, 2014

Supervisor

Professor Sokratis K. Katsikas,
University of Piraeus

Examination Board

CONTENTS

Acknowledgements

This project would not have been possible without the existence and maintenance of the Linux from Scratch Project. I would also like to thank my supervisor Professor Sokratis Katsikas for his guidance and insights.

Piraeus, February 2014

Abstract

The various forms of UNIX inherently suffer from security issues. However, forms of the Unix OS family are widely used, often in critical applications. The aim of this work is to present the process of developing a new operating system, based on Linux. The system will be developed from the outset from source code, not based on any existing distribution and will emphasize on the shielding of the entire system and its individual parts in order to provide better security and be able to accommodate critical (military grade) applications of interest.

This thesis presents the process of creating a secure OS in two stages:

At a first stage, all the steps necessary to construct a brand new open source operating system based on Linux are followed. To achieve this goal, the Linux From Scratch (LFS) project is utilized [1]. The outcome is a new basic Linux operating system with the minimum amount of software needed to operate and is based on the Linux 3.10.10 Kernel.

At a second stage, security measures to harden the security of the new OS are explored. All major Linux Security Modules (LSM) and security suites are presented and a framework for their comparison based on common attributes is suggested. Using this framework, the Security Enhanced Linux (SELinux) project is chosen to harden the new OS. The suite is then more thoroughly presented and applied. Other security measures for further hardening our new system are presented and applied when possible. As a last step, we explore other possibilities for evolving our security hardened OS implementation.

Chapter 1

Introduction

Current information technology status reveals a rather intimidating canvas security-wise: Recent claims suggest the involvement of major software and operating system vendors in information gathering by security agencies [2]. Various techniques of placing back-doors on operating systems and devices for information gathering where suggested. The story goes as back as 1999, when the "_NSAKEY" cryptographic variable was discovered in Windows NT, with many security experts suggesting that it enabled a back-door on Microsoft's Windows 2000 OS[3]. Recently, Germany's Office for Information Security (BSI) expressed concerns on the Trusted Platform Module technology used in Windows 8 in conjunction with the Trusted Platform Module (TPM) 2.0 chip [4].

Claims of condescending assistance to intelligence agencies for information gathering have been refuted by software and operating system developers. With the extent of operating systems compromise unknown, a new need presents itself: The need for a trusted OS designed with security first in mind. This operating system should be able provide security as a fundamental function for critical and military grade applications. This is an effort to suggest such a system.

1.1 Problem Definition

The Linux operating system is assembled under the model of free and open source software development and distribution (FOSS) [5]. As it is free to use and stable at its performance, it has become the leading OS for servers, mainframes and supercomputers. It is built into the firmware of most embedded systems. Even the most popular operating system on mobile devices, the Android OS is based on Linux. Linux adoption in the enterprise has been accelerating for more than a decade [6].

While thought of as a generally secure operating system, Linux is based on UNIX which was not designed with security first in mind. Inherently insecure services such as ftp and telnet can be potential paths for entrance of malicious users into the system. Exploitable bugs in server software our found almost at a daily basis and if not properly and timely patched, they can lead to unauthorized access and system exploitation [7].

In order to be used as a container in critical or sensitive (military grade) applications, we need to take all necessary steps and measures to harden Linux from a security point of view.

1.2 Thesis Structure

The problem is addressed in stages:

● First, a brand new Linux system is constructed by compiling all packets from source code.

● Our new Linux OS is then hardened by applying specific security measures. While writing this thesis, it became imperative that a framework of the existing security enhancement solutions was introduced in order to choose from.

● To harden even more, other popular security measures are analyzed and applied if possible.

● Finally, ways of continuing the work presented are suggested.

1.3 Contribution

This thesis contributes in increasing the security of the Linux OS in various ways:

● It describes and practices the procedure of constructing of a new Linux OS. The created OS is a prototype, a basic system to build up from, selecting and installing only the software packages needed for a specific task. This minimizes, in the most efficient way possible, the installed software fingerprint and thus provides the smallest possible surface for finding and exploiting software bugs.

● It proposes a framework for comparing the existing projects for Linux security hardening. This helps the reader to decide which way to go, applying the solution or the combination of solutions most suitable for the enterprise needs.

● By presenting up-to-date security measures, it can be used as a security guide from system administrators and help them better secure their systems.

● By combining a purpose oriented OS with a minimal software fingerprint and the appropriate security measures for its function, one can produce highly secure specialized government or military systems, suitable for critical and sensitive applications.

Chapter 2

Creating a New Operating System

2.1 Introduction

In order to create a new Linux operating system from the source code up, this thesis utilizes the Linux From Scratch Project [1]. Linux From Scratch (from now on referred to as LFS) is a project that provides step-by-step instructions for building a custom Linux system entirely from source code.

LFS follows Linux standards as closely as possible. The primary standards are:

- POSIX.1-2008 [8]
- Filesystem Hierarchy Standard (FHS)[9]
- Linux Standard Base (LSB) Specifications [10]

The steps and procedures described in the latest LFS book release, version 7.4, where implemented. The outcome is a Linux System based on the 3.10.10 kernel.

2.2 Benefits of starting from scratch

One could argue that there is no reason for building a new Linux operating system from the beginning. There are already many distributions out there, some supporting advanced security features that one could use as a secure application container. While choosing an already existing distribution is a valid option, there are certain benefits in starting the building process from the beginning:

- Building an operating system from source code provides the knowledge of how a Linux system works internally, what are the dependencies between different programs and how they interact in the creating process. This knowledge is crucial in understanding how the system operates and how to better secure it.

- A complete LFS system provides a very basic working system, a skeleton upon which to build. Following the LFS process can produce minimal, purpose driven systems that include only the absolutely necessary software packets to operate. This is a great benefit as it reduces the software base and consecutively the amount of vulnerabilities in the operating system.

- Compiling the entire system from source code means that every aspect of the operating system can be reviewed and if necessary audited for security purposes. If a security hole is discovered, it can be patched without the need for a third party to release a security patch.

2.3 Building the LFS System

The system was built on a dual-core laptop with Ubuntu Linux 12.04 32 bit version as its operating system (from now on referred to as **host**). The steps described in the LFS book up to chapter 9 where followed, where the new OS is considered complete and can be chrooted in. As the system was developed in an older laptop system with a minimal hardware and software configuration, a bootable version of the produced OS will not be presented and the extra steps necessary to make the system bootable in the specific host system are considered out of scope.

As detailed step by step instructions for the build are provided in the LFS book, the walkthrough of the creation process will be kept at a higher, more explanatory level, focusing on key concepts at every step of the build.

2.3.1. Process Overview

In order to build a new system, an already installed Linux distribution such as Debian, Mandriva, Red Hat, or SUSE is required. This host operating system will provide the necessary programs, including a compiler, linker, and shell, to build the new system. After a new partition and file system are prepared, a number of packages are downloaded into it, to build the basic development suite (from now on referred to as **toolchain**).

We then proceed to build a first pass of the toolchain, including Binutils and GCC (first pass basically means these two core packages will be reinstalled). The next step is to build Glibc, the C library. Glibc will be compiled by the toolchain programs built in the first pass. Then, a second pass of the toolchain will be built. This time, the toolchain will be dynamically linked against the newly built Glibc. The remaining packages are built using this second pass toolchain. The purpose of this operation is to insure that the LFS installation process will no longer depend on the host distribution.

Finally, the full system is built. All packages that will constitute our new OS are compiled using the toolchain created in the previous phase. Bootscripts and the Linux Kernel are set up. The **chroot** (change root) program is used to enter a virtual environment and start a new shell whose root directory will be set to the new system partition. This is similar to rebooting and instructing the kernel to mount the LFS partition as the root partition. The system does not actually reboot, but instead **chroots** into the new system.

2.3.2. Preparation

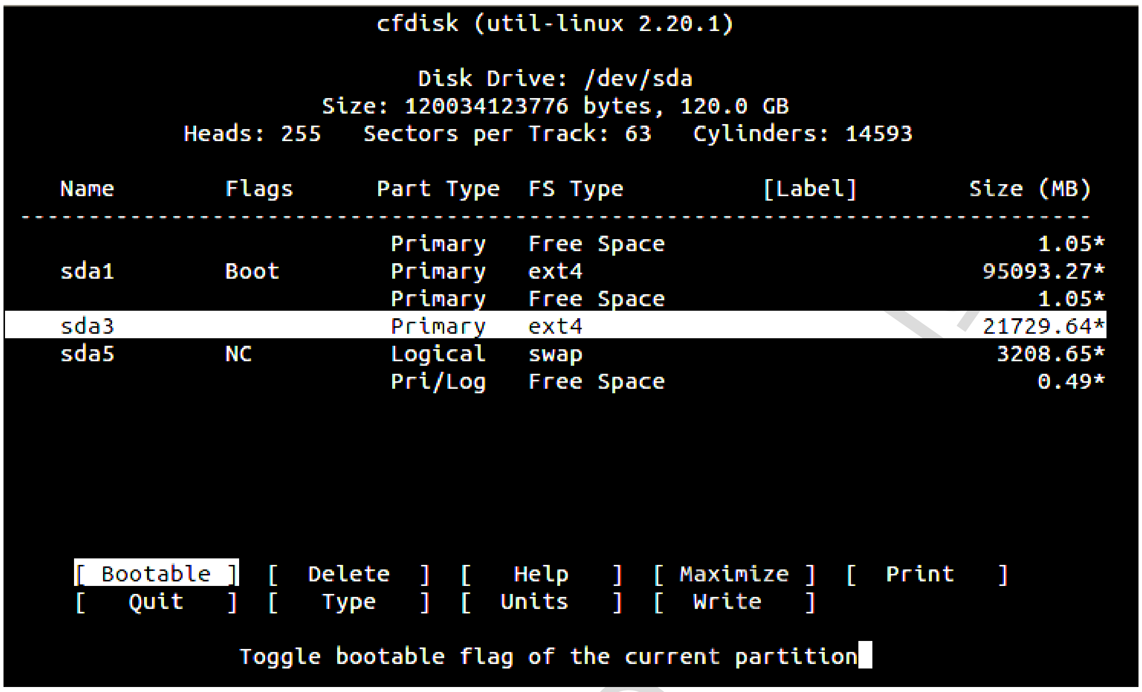
The following script checks and reports if all host system requirements are met:

```
cat > version-check.sh << "EOF"
#!/bin/bash
# Simple script to list version numbers of critical development tools

export LC_ALL=C
bash --version | head -n1 | cut -d" " -f2-4
echo "/bin/sh -> `readlink -f /bin/sh`"
echo -n "Binutils: "; ld --version | head -n1 | cut -d" " -f3-
bison --version | head -n1
```

```
if [ -e /usr/bin/yacc ];
  then echo "/usr/bin/yacc -> `readlink -f /usr/bin/yacc`";
  else echo "yacc not found"; fi

bzip2 --version 2>&1 < /dev/null | head -n1 | cut -d" " -f1,6-
echo -n "Coreutils: "; chown --version | head -n1 | cut -d")" -f2
diff --version | head -n1
find --version | head -n1
gawk --version | head -n1
if [ -e /usr/bin/awk ];
  then echo "/usr/bin/awk -> `readlink -f /usr/bin/awk`";
  else echo "awk not found"; fi

gcc --version | head -n1
g++ --version | head -n1
ldd --version | head -n1 | cut -d" " -f2-  # glibc version
grep --version | head -n1
gzip --version | head -n1
cat /proc/version
m4 --version | head -n1
make --version | head -n1
patch --version | head -n1

echo Perl `perl -V:version`
sed --version | head -n1
tar --version | head -n1
echo "Texinfo: `makeinfo --version | head -n1`"
xz --version | head -n1

echo 'main(){}' > dummy.c && g++ -o dummy dummy.c
if [ -x dummy ]
  then echo "g++ compilation OK";
  else echo "g++ compilation failed"; fi
rm -f dummy.c dummy
EOF
bash version-check.sh
```

In our Ubuntu host only 3 programs had to be installed: **bison**, **gawk** and **texinfo**.

Using a disk partitioning program, a new 20GB partition was created on the computer’s hard disk. The newly created partition was formatted for the ext4 file system:



```
                          cfdisk (util-linux 2.20.1)

                              Disk Drive: /dev/sda
                       Size: 120034123776 bytes, 120.0 GB
             Heads: 255   Sectors per Track: 63   Cylinders: 14593

    Name        Flags      Part Type  FS Type          [Label]        Size (MB)
 ------------------------------------------------------------------------------
                            Primary   Free Space                          1.05*
    sda1        Boot        Primary   ext4                            95093.27*
                            Primary   Free Space                          1.05*
    sda3                    Primary   ext4                            21729.64*
    sda5        NC          Logical   swap                             3208.65*
                            Pri/Log   Free Space                          0.49*




     [ Bootable ]  [  Delete  ]  [   Help   ]  [ Maximize ]  [  Print   ]
     [   Quit   ]  [   Type   ]  [  Units   ]  [  Write   ]

                  Toggle bootable flag of the current partition
```

Image 2.1: An ext4 partition (sda3) was prepared on the host disk.

Throughout the build, the environment variable **$LFS** will be used and must always be defined. Next we mount the newly created partition and create a folder to put downloaded software. We also create a /tools folder. This is the folder where all compiled programs will be installed:

```
#Define the LFS variable:
export LFS=/mnt/lfs
mkdir -pv $LFS
mount -v -t ext3 /dev/sda3 $LFS

#Create the sources folder and download software:
mkdir -v $LFS/sources
chmod -v a+wt $LFS/sources
wget -i wget-list -P $LFS/sources

#Create the tools folder:
mkdir -v $LFS/tools
ln -sv $LFS/tools /
```

2.3.3. Software Packages

To build a basic Linux system, the following software packages are needed and where downloaded into $LFS/sources. All software signatures where verified using the **md5sums** program. In addition to these packages, a number of patches where downloaded to address compatibility issues and provide package-specific fixes. Each program's functionality is described in **Appendix A**

List of Software Packages		
1. Autoconf (2.69)	22. GMP (5.1.2)	43. Patch (2.7.1)
2. Automake (1.14)	23. Grep (2.14)	44. Perl (5.18.1)
3. Bash (4.2)	24. Groff (1.22.2)	45. Pkg-config (0.28)
4. Bc (1.06.95)	25. GRUB (2.00)	46. Procps (3.3.8)
5. Binutils (2.23.2)	26. Gzip (1.6)	47. Psmisc (22.20)
6. Bison (3.0)	27. Iana-Etc (2.30)	48. Readline (6.2)
7. Bzip2 (1.0.6)	28. Inetutils (1.9.1)	49. Sed (4.2.2)
8. Check (0.9.10)	29. IPRoute2 (3.10.0)	50. Shadow (4.1.5.1)
9. Coreutils (8.21)	30. Kbd (1.15.5)	51. Sysklogd (1.5)
10. DejaGNU (1.5.1)	31. Kmod (14)	52. Sysvinit (2.88dsf)
11. Diffutils (3.3)	32. Less (458)	53. Tar (1.26)
12. E2fsprogs (1.42.8)	33. LFS-Bootscripts (20130821)	54. Tcl (8.6.0)
13. Expect (5.45)	34. Libpipeline (1.2.4)	55. Time Zone Data (2013d)
14. File (5.14)	35. Libtool (2.4.2)	56. Texinfo (5.1)
15. Findutils (4.4.2)	36. Linux (3.10.10)	57. Systemd (206)
16. Flex (2.5.37)	37. Make (3.82)	58. Udev-lfs Tarball (206)
17. Gawk (4.1.0)	38. Man-DB (2.6.5)	59. Util-linux (2.23.2)
18. GCC (4.8.1)	39. Man-pages (3.53)	60. Vim (7.4)
19. GDBM (1.10)	40. MPC (1.0.1)	61. Xz Utils (5.0.5)
20. Gettext (0.18.3)	41. MPFR (3.1.2)	62. Zlib (1.2.8)
21. Glibc (2.18)	42. Ncurses (5.9)	
List of Software Patches		
1. Automake Test Fix Patch		
2. Bash Upstream Fixes Patch		
3. Bzip2 Documentation Patch		
4. Coreutils Internationalization Fixes Patch		
5. Kbd Backspace/Delete Fix Patch		
6. Make Upstream Fixes Patch		
7. Perl Libc Patch		
8. Tar Manpage Patch		
9. Readline Upstream Fixes Patch		
10. Texinfo Test Patch		

Table 2.1: Software Packages and Patches used in the LFS build

2.3.4. Constructing a Temporary System

The first step in building our OS is to build a new and host-independent toolchain (compiler, assembler, linker, libraries, and a few useful utilities). This step is necessary to ensure a clean, trouble-free build of the target system:

First, the LFS user is assigned proper rights to the LFS folders:

```
groupadd lfs
useradd -s /bin/bash -g lfs -m -k /dev/null lfs
passwd lfs
chown -v lfs $LFS/tools
```

The build process is as follows:

For each package:

- Using the **tar** program, extract the package to be built.
- Change to the directory created when the package was extracted.
- Build the package using the configure, make, make install commands as specifically instructed for each individual packet. Software in installed in the tools directory.
- Change back to the sources directory.
- Delete the extracted source directory and any <package>-build directories that were created in the build process.

In strict order, these packages where compiled:

1. Binutils-2.23.2 - Pass 1	16. Diffutils-3.3
2. GCC-4.8.1 - Pass 1	17. File-5.14
3. Linux-3.10.10 API Headers	18. Findutils-4.4.2
	19. Gawk-4.1.0
4. Glibc-2.18	20. Gettext-0.18.3
5. Libstdc++-4.8.1	21. Grep-2.14
6. Binutils-2.23.2 - Pass 2	22. Gzip-1.6
7. GCC-4.8.1 - Pass 2	23. M4-1.4.16
8. Tcl-8.6.0	24. Make-3.82
9. Expect-5.45	25. Patch-2.7.1
10. DejaGNU-1.5.1	26. Perl-5.18.1
11. Check-0.9.10	27. Sed-4.2.2
12. Ncurses-5.9	28. Tar-1.26
13. Bash-4.2	29. Texinfo-5.1
14. Bzip2-1.0.6	30. Xz-5.0.5
15. Coreutils-8.21	

Table 2.2: Packages used to construct the Toolchain

At the end of the process the toolchain is complete. We now have a minimal set of tools that are necessary to build our final system. The time it takes to compile the toolchain can be significant and depends greatly on the system’s processor and the lab configuration.

2.3.5. Compiling the Basic System Software

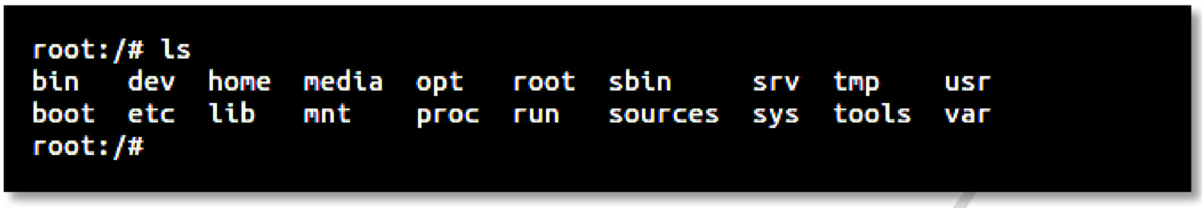
With the toolchain complete, we are ready to enter the build environment and start shaping our new system:

```
chroot "$LFS" /tools/bin/env -i \
    HOME=/root                  \
    TERM="$TERM"                \
    PS1='\u:\w\$ '              \
    PATH=/bin:/usr/bin:/sbin:/usr/sbin:/tools/bin \
    /tools/bin/bash --login +h
```

We create a standard Linux directory tree and all necessary compatibility symlinks for the man, doc, and info directories by issuing:

```
mkdir -pv /{bin,boot,etc/{opt,sysconfig},home,lib,mnt,opt,run}
mkdir -pv /{media/{floppy,cdrom},sbin,srv,var}
install -dv -m 0750 /root
install -dv -m 1777 /tmp /var/tmp
mkdir -pv /usr/{,local/}{bin,include,lib,sbin,src}
mkdir -pv /usr/{,local/}share/{doc,info,locale,man}
mkdir -v  /usr/{,local/}share/{misc,terminfo,zoneinfo}
mkdir -pv /usr/{,local/}share/man/man{1..8}
for dir in /usr /usr/local; do
  ln -sv share/{man,doc,info} $dir
done
case $(uname -m) in
 x86_64) ln -sv lib /lib64 && ln -sv lib /usr/lib64 && ln -sv
lib /usr/local/lib64 ;;
esac
mkdir -v /var/{log,mail,spool}
ln -sv /run /var/run
ln -sv /run/lock /var/lock
mkdir -pv /var/{opt,cache,lib/{misc,locate},local}
```

At this point we have the final directory structure of our new OS:



```
root:/# ls
bin    dev  home  media  opt   root  sbin     srv  tmp    usr
boot   etc  lib   mnt    proc  run   sources  sys  tools  var
root:/#
```

Image 2.2: Directory Structure of the Final System

We can now start compiling the final system. Following the same steps used in section 2.3.4 to build the toolchain, the software packages where compiled in the exact following order:

1. Linux-3.10.10 Headers	21. Coreutils-8.21	41. Xz-5.0.5
2. Man-pages-3.53	22. Iana-Etc-2.30	42. GRUB-2.00
3. Glibc-2.18	23. M4-1.4.16	43. Less-458
4. Adjusting the Toolchain	24. Flex-2.5.37	44. Gzip-1.6
5. Zlib-1.2.8	25. Bison-3.0	45. IPRoute2-3.10.0
6. File-5.14	26. Grep-2.14	46. Kbd-1.15.5
7. Binutils-2.23.2	27. Readline-6.2	47. Kmod-14
8. GMP-5.1.2	28. Bash-4.2	48. Libpipeline-1.2.4
9. MPFR-3.1.2	29. Bc-1.06.95	49. Make-3.82
10. MPC-1.0.1	30. Libtool-2.4.2	50. Man-DB-2.6.5
11. GCC-4.8.1	31. GDBM-1.10	51. Patch-2.7.1
12. Sed-4.2.2	32. Inetutils-1.9.1	52. Sysklogd-1.5
13. Bzip2-1.0.6	33. Perl-5.18.1	53. Sysvinit-2.88dsf
14. Pkg-config-0.28	34. Autoconf-2.69	54. Tar-1.26
15. Ncurses-5.9	35. Automake-1.14	55. Texinfo-5.1
16. Shadow-4.1.5.1	36. Diffutils-3.3	56. Udev-206 (Extracted from systemd-206)
17. Util-linux-2.23.2	37. Gawk-4.1.0	57. Vim-7.4
18. Psmisc-22.20	38. Findutils-4.4.2	
19. Procps-ng-3.3.8	39. Gettext-0.18.3	
20. E2fsprogs-1.42.8	40. Groff-1.22.2	

Table 2.3: Installed Software Packages in compilation order

When the time-consuming process of compiling each individual packet is complete, the newly created system has a minimum set of tools and programs to operate as a basic but complete standalone OS. Only two more steps are required, setting up the bootscripts and, finally compiling the Linux Kernel itself.

2.3.6. Setting up Bootscripts

Proper system boot process and operation requires the installation of a number of scripts and configuration files. A listing of the boot scripts and configuration files that where installed are found in [11].

2.3.7. Installation of the Linux Kernel

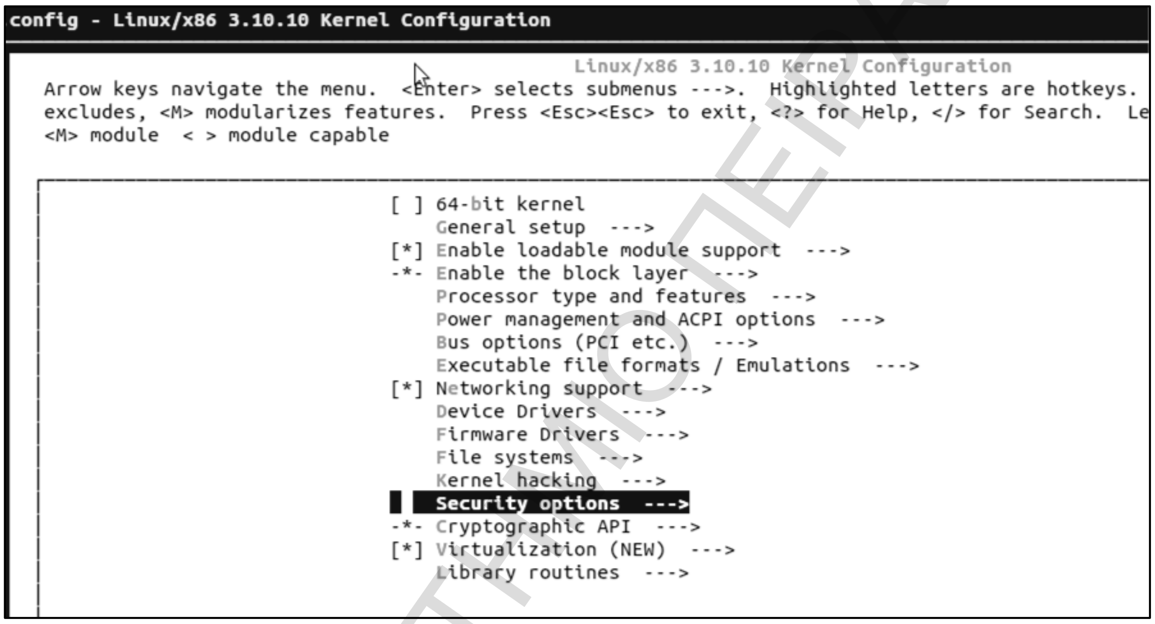
This is the core of the new operating system and the final step in this exercise. Building the kernel involves three steps: Configuration, compilation, and installation.

● Configuration: Prior to compiling the kernel tree has to be clean. This is accomplished by issuing the following command:

```
make mrproper
```

The kernel will then be configured via a menu driven interface, by issuing:

```
make menuconfig
```



```
config - Linux/x86 3.10.10 Kernel Configuration
                                  Linux/x86 3.10.10 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.
  excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Le
  <M> module  < > module capable

                        [ ] 64-bit kernel
                            General setup  --->
                        [*] Enable loadable module support  --->
                        -*- Enable the block layer  --->
                            Processor type and features  --->
                            Power management and ACPI options  --->
                            Bus options (PCI etc.)  --->
                            Executable file formats / Emulations  --->
                        [*] Networking support  --->
                            Device Drivers  --->
                            Firmware Drivers  --->
                            File systems  --->
                            Kernel hacking  --->
                            Security options  --->
                        -*- Cryptographic API  --->
                        [*] Virtualization (NEW)  --->
                            Library routines  --->
```

Image 2.3: Linux/x86 3.10.10 Kernel Configuration Options

This brings up a menu driven utility where all core system configuration are setup. For this exercise, we will work with the default configuration on all options except security and some file systems options, which will be re-examined in Chapter 4.

Compilation and installation of the kernel after configuration has been complete involves the following:

● Compile the kernel image and selected modules:

```
make
make_modules install
```

● The kernel image, system map and config files are moved to the /boot directory:

```
cp -v arch/x86/boot/bzImage /boot/vmlinuz-3.10.10-lfs-7.4
cp -v System.map /boot/System.map-3.10.10
cp -v .config /boot/config-3.10.10
```

2.4 Final Result

Our brand new operating system is now complete. If the build was made on a separate hard disk we could boot from it. Due to the specific hardware configuration of the host computer, a bootable version of the final result, will not be presented. With the exception of running processes, the system can be chrooted in and commands executed. On the next chapters, ways to provide better security against common attacks are discussed and specific security measures are suggested.

Chapter 3

Linux Security Features

3.1 Introduction

"The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes." [12]

Linux started out with the traditional Unix security model, **DAC** (Discretionary Access Control model). DAC allows the owner of an object (such as a file) to set the security policy for that object which is why it's called a discretionary scheme. While simple and still effective, it does not meet the modern internet era security needs. For example, exploiting buggy software an attacker can gain access with the rights of the user, which can be very dangerous if that user is the super user (root).

Over the time Linux security involved, introducing more advanced security mechanisms to protect the OS, such as **MAC** (Mandatory Access Control) or **RBAC** (Role Based Access Control) that will discussed in the next session. Several different approaches and different methodology to security have been presented and are incorporated in the kernel as modules. These are the **LSM** (Linux Security Modules). It must be noted that only one LSM can be active in the Kernel.

3.2 Linux Security Modules

The LSM API adds hooks at all security-critical points within the kernel. The implemented LSM receives callbacks from these hooks and may deny the operation. It also allows different security models to be passed into the Kernel, such as ACL's. DAC checks are always performed first and only if they succeed the LSM control is invoked.

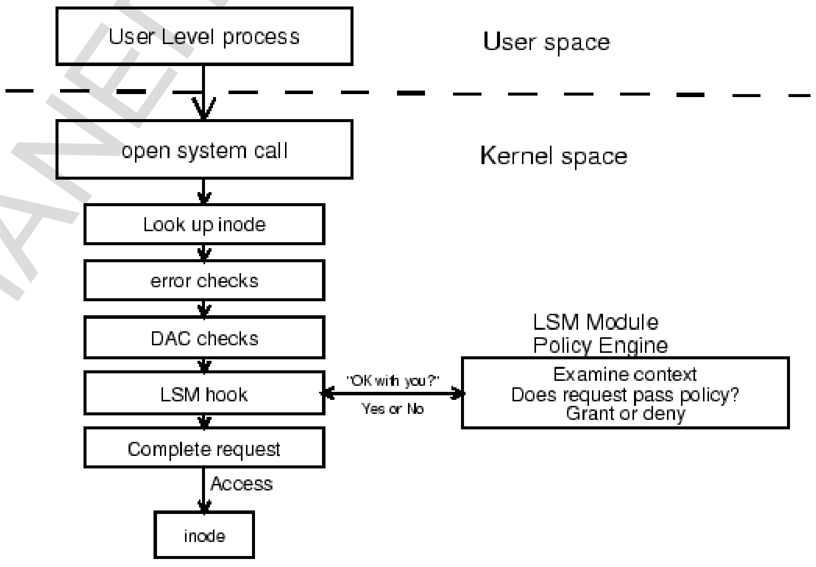


Image 3.1: LSM hook mechanism

3.2.1. SELinux

SELinux (Security Enhanced Linux) was developed by the NSA (United States National Security Agency)[13][14] and is now a community supported open source project. It has been implemented in the kernel since the 2.6 version. It provides a flexible MAC scheme called **FLASK** to a meet a wide variety of security requirements, from general purpose use to government and military systems. MAC differs from DAC in that the security policy is administered centrally and not by the user. This means that the user does not administer the security policy for his own resources. This greatly helps against attacks on exploitable software bugs or wrong configurations. All objects on the system are assigned security labels and all interactions are hooked and passed to the SELinux LSM module which allows or denies it according to the security policy. The security policy can be fine-grained to meet specific security goals.

SELinux is a mature project and is implemented as a standard feature amongst others in the Red Hat commercial, Fedora and CentOS Linux distributions.

3.2.2. AppArmor

AppArmor was developed by Immunix and is since 2009 maintained by Canonical [15]. Designed with ease of use in mind, it applies MAC controls to applications. Its main difference with SELinux is that it uses pathnames to applications instead of their inodes (object names) to apply the security policy. AppArmor policies are called profiles and work by setting the abilities and file permissions of a program. AppArmor can automatically generate profiles by running it in a special learning mode.

AppArmor ships with the Ubuntu and OpenSUSE Linux distributions.

3.2.3. Smack

Smack (Simplified Mandatory Access Control Kernel) LSM also uses a MAC framework. Its goal is to provide good security without the complexity of SELinux policies. It uses extended attributes (xattrs) to store labels on filesystem objects and the security policy can also be modified. There are 4 predefined labels used to enforce access permission based on a simple set of rules[16].

Smack is part of the Tizen Project [17].

3.2.4. TOMOYO

TOMOYO LSM (Task Oriented Management Obviates Your Onus) is a path-based MAC module [18] that it is aimed at simplicity. It uses process execution history and has a learning mode just like AppArmor. TOMOYO names the paths as domains. A domain records a chain of tasks from the boot-up to a specific program execution and is used for policy installment. TOMOYO is considered rather user than administrator oriented and is low on adoption.

3.2.5. YAMA

YAMA [19] is a new standard LSM. It uses a DAC scheme to apply security enhancements mostly inspired from the grsecurity project. It currently implements enhanced restrictions on ptrace, to prevent an attacker from compromising an application and use it to attach to other running processes. In specific modes of operation, YAMA can be stacked with other security modules when used as a kernel patch.

3.3 grsecurity

Grsecurity is a set of patches for the Linux kernel with an emphasis on enhancing security. It includes Access Control Lists (**ACL**) to allow the system administrator to define a least privilege policy for the system, in which every process and user have only the lowest privileges needed to function. Grsecurity has not been ported as a security module for the kernel, due to its developer disagreement with the LSM model [20].

Despite the fact that it is not implemented as a LSM, grsecurity is worth mentioning as it offers significant security enhancements like the Pax Project, that efficiently protect a Linux system mainly against memory smashing exploitation techniques and network-based exploitations.

3.4 Comparison of Security Frameworks

To apply core security measures, one of the available security frameworks has to be selected. To help decide on which security solution better meets our needs, a set of criteria will be set to compare them. Official documentation as well as [21][22][23] where utilized to complete our set of criteria:

i. LSM implementation: Kernel features can be applied either as patches or as modules. Whether a security suite is applied as a patch or as a module is important, as official LSM modules are included as part of the mainline kernel releases and are fully compatible with it.

ii. Object Selection: The method used by the framework to assign security labels to objects. Possible options are inode, filename, process tree. The method is important as using inodes results in a larger number of supported objects than the other two methods (with a cost on policy complexity).

iii. Supported Objects: A derivative of the object selection method, it depicts the kind of objects (files, filesystem, processes, actions, sockets). The more objects a framework supports, the better.

iv. Automatic Policy Generation: Whether the framework has the ability to produce policies automatically by monitoring object behavior or not.

v. Memory Protection: Whether the framework provides security against memory exploitation attacks.

vi. Whitelisting: Whether the framework provides the ability to add exceptions to the enforced security policy.

vii. Implementation complexity: Based on the framework's documentation, installation procedure and policy generation techniques, each suite will be given a rank of either high or low.

viii. Security certifications: The existence of security certifications, specifically for the security framework as a whole or as part of a platform is considered a great advantage.

vii. Popularity: Based on the security framework's adoption in modern popular Linux distributions, support from the security community and bibliography, each suite will be given a rank of either high or low.

Name	SELinux	AppArmor	SMACK	TOMOYO	YAMA	grsecurity
LSM implementation	Yes	Yes	Yes	Yes	Yes	No
Object Selection	inode	filename	inode (follow symlinks)	process tree	filename	filename
Supported Objects	• files • filesystem • actions • sockets • processes	• files • actions • sockets	• files • processes • sockets	files	processes	• files • actions • sockets • resources
Automatic Policy Generation	No	Yes	No	Yes	No	Yes
Memory Protection	• heap • stack	No	No	No	No	PaX
Whitelisting	Yes	No	Yes	Yes	Yes	Yes
Implementation complexity	High	Low	Low	Low	Low	High
Security certifications	• EAL4+ • CAPP • LSPP[ref]	No	No	No	No	No
Popularity	High	High	Low	Low	Low	High

Table 3.4: Attribute values for the evaluated security frameworks

3.5 Selection

Our security framework of choice is **SELinux**. Developed by a security aware organization, it is adopted as a standard in Red Hat, CentOS and Fedora Linux distributions. As a security module it is part of all mainstream Kernel releases and although it is difficult to fine grain and properly tune, it supports training, security certifications and protects the widest range of objects of a Linux OS.

Grsecurity, despite its advanced features mostly against memory exploitations, falls short due to the absence of a security module and complexity of implementation. AppArmor, our second runner up, works well in application firewalling but does not protect the system as a whole nor it supports the number of objects supported by SELinux.

One point to make here is that this simple comparison framework can result in choosing a different security suite if for example a user seeks the best way to protect his personal Linux laptop. Our goal is to select the security suite that, complexity aside, can maximize our new OS security.

Chapter 4

Implementing SELinux

4.1 Kernel SELinux Integration

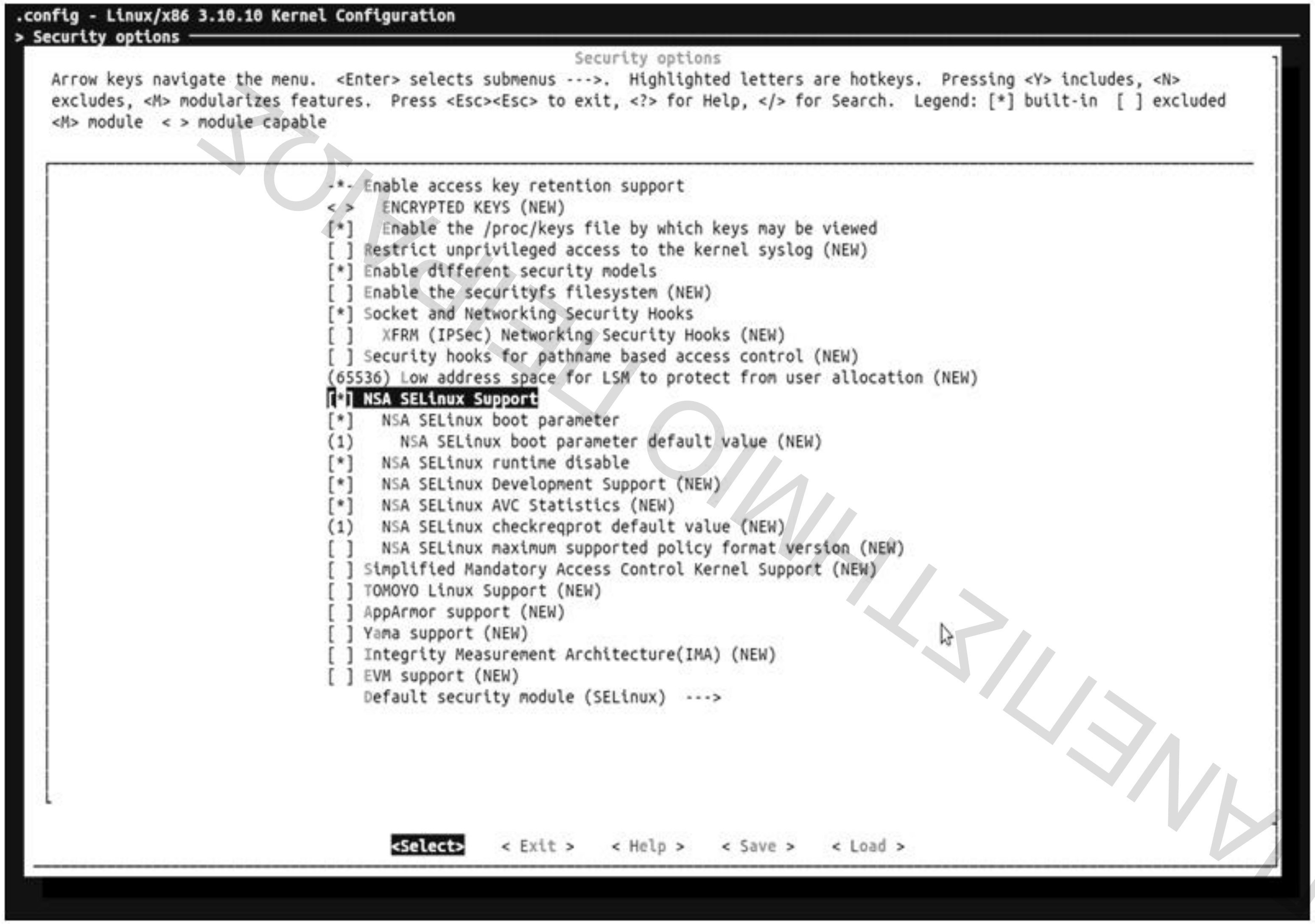
In this section the SELinux kernel options, module integration and the final kernel compilation for our new system are presented. In order to activate the SELinux module the entire kernel has to be recompiled. Again, we issue:

```
make mrproper
make menuconfig
```

This will clean all previous configurations and open the kernel menu driven interface. In order to attach extended security attributes to objects, we check:

- File systems -> Ext3 extended attributes and Ext3 Security Labels
- File systems -> Ext4 extended attributes and Ext4 Security Labels

We can now proceed with the main security configuration of the kernel, in order to support the SELinux Security Module [27]. **Image 4.1** shows all available security options in the 3.10.10 kernel:



```
.config - Linux/x86 3.10.10 Kernel Configuration
> Security options
                                  Security options
 Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> includes, <N>
 excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
 <M> module  < > module capable

                 -*- Enable access key retention support
                 < >   ENCRYPTED KEYS (NEW)
                 [*]   Enable the /proc/keys file by which keys may be viewed
                 [ ] Restrict unprivileged access to the kernel syslog (NEW)
                 [*] Enable different security models
                 [ ] Enable the securityfs filesystem (NEW)
                 [*] Socket and Networking Security Hooks
                 [ ]   XFRM (IPSec) Networking Security Hooks (NEW)
                 [ ] Security hooks for pathname based access control (NEW)
                 (65536) Low address space for LSM to protect from user allocation (NEW)
                 [*] NSA SELinux Support
                 [*]   NSA SELinux boot parameter
                 (1)     NSA SELinux boot parameter default value (NEW)
                 [*]   NSA SELinux runtime disable
                 [*]   NSA SELinux Development Support (NEW)
                 [*]   NSA SELinux AVC Statistics (NEW)
                 (1)   NSA SELinux checkreqprot default value (NEW)
                 [ ]   NSA SELinux maximum supported policy format version (NEW)
                 [ ] Simplified Mandatory Access Control Kernel Support (NEW)
                 [ ] TOMOYO Linux Support (NEW)
                 [ ] AppArmor support (NEW)
                 [ ] Yama support (NEW)
                 [ ] Integrity Measurement Architecture(IMA) (NEW)
                 [ ] EVM support (NEW)
                     Default security module (SELinux)  --->

                    <Select>    < Exit >    < Help >    < Save >    < Load >
```

Image 4.1: KernelSecurity Configuration Options

● **NSA SELinux Support**

This enables SELinux. To be able to apply security policies, we will also need a policy configuration and a labeled filesystem.

● **NSA SELinux boot parameter/ boot parameter default value**

This option adds a kernel parameter 'selinux', which allows SELinux to be disabled at boot. If this option is set to 1, the SELinux kernel parameter will default to 1, enabling SELinux at bootup. Our choice here will be **1**.

● **NSA SELinux runtime disable**

This option enables writing to a selinuxfs node 'disable', which allows SELinux to be disabled at runtime prior to the policy load. SELinux will then remain disabled until the next boot. This option supports runtime disabling of SELinux, for portability across platforms where boot parameters are difficult to employ. This option will be **checked**.

● **NSA SELinux Development Support**

This enables the development support option of NSA SELinux, which is useful for experimenting with SELinux and developing policies. With this option enabled, the kernel will start in permissive mode (log everything, deny nothing) unless we specify enforcing=1 on the kernel command line. For our purpose this is a required feature and will be **checked**.

● **NSA SELinux AVC Statistics**

This option collects AVC (Access Vector Cache) statistics to /selinux/avc/cache_stats, which may be monitored via tools such as avcstat. This is not a critical feature and will be **not checked**.

● **NSA SELinux checkreqprot default value**

This option sets the default value for the 'checkreqprot' flag that determines whether SELinux checks the protection requested by the application or the protection that will be applied by the kernel. If this option is set to 0 (zero), SELinux will default to checking the protection that will be applied by the kernel. If this option is set to 1 (one), SELinux will default to checking the protection requested by the application. Our choice here will be **1**.

● **NSA SELinux maximum supported policy format version/value**

This option enables the maximum policy format version supported by SELinux to be set to a particular value. This mostly refers to Fedora-based distributions and will be **not checked**.

With all choices regarding general and SELinux configuration made, we are now ready to re-compile the Kernel:

```
make
make_modules install
```

● Again, the new kernel image, system map and config files have to be moved to the /boot directory (previous versions must be deleted):

```
cp -v arch/x86/boot/bzImage /boot/vmlinuz-3.10.10-lfs-7.4
cp -v System.map /boot/System.map-3.10.10
cp -v .config /boot/config-3.10.10
```

4.2 Userspace Configuration

To manage SELinux users, roles and policies, a number of software packages have to be obtained and installed: **libselinux**, **libsemanage**, **libsepol**, **policycoreutils** and **sepolgen**. In addition, a number of SELinux aware programs can replace standard ones : cp, find, id, ls, mkdir are amongst them. Currently the packages are maintained and can be downloaded from the selinuxproject software repository [24].

During the next boot, the entire file system is labeled. The labeling process labels all files with a SELinux context:

```
*** Warning -- SELinux targeted policy relabel is required.
*** Relabeling could take a very long time, depending on file system
size and speed of hard drives.
****
```

This is a sample output of a ps command on a SELinux enabled system:

```
[root@~ /]# ps -e --context
      PID   SID CONTEXT                         COMMAND
        1     7 system_u:system_r:init_t        init
        2     1 system_u:system_r:kernel_t      [keventd]
        3     1 system_u:system_r:kernel_t      [kapmd]
        4     1 system_u:system_r:kernel_t      [ksoftirqd_CPU0]
        5     1 system_u:system_r:kernel_t      [kswapd]
        6     1 system_u:system_r:kernel_t      [bdflush]
        7     1 system_u:system_r:kernel_t      [kupdated]
      515   274 system_u:system_r:syslogd_t     syslogd -m 0
     2726   297 system_u:system_r:sshd_t        /usr/sbin/sshd
     2728   323 root:user_r:user_t              -bash
     2920   323 root:user_r:user_t              ps -e --context
    [root@~ /]#
```

We see two extra columns giving the SID (Security IDentifier tag) and the associated **user**, **role**, and **type** for the process being displayed.

4.3 The SELinux Policy

The CONTEXT column in the ps output above in known in SELinux as the **Security Context**: It is made of three elements, the **identity**, the **role** and the **domain** or **type**. The identity can have the same name but is not the same as the Unix uid. Every process runs in a domain which determines the access a process has. A type is assigned to an object and determines who gets to access that object. The difference is that a domain applies to process and a type applies to objects such as directories, files and sockets. A role determines what domains can be used. The domains that a user role can access are predefined in policy configuration files. If a role is not authorized to enter a domain (in the policy database), then it will be denied.

As an example running *ls --context /proc* in a SELinux enabled system shows the following listing for the init process:

```
[root@~ /]# ps -e --context
dr-xr-xr-x  root     root     system_u:system_r:init_t
[root@~ /]#
```

Here, the init process (domain) runs under the system_u (user) with the system_r (role). Policy is the set of rules that guide the SELinux security engine. It defines types for file objects and domains for processes, uses roles to limit the domains that can be entered, and has user identities to specify the roles that can be attained[26].

SELinux follows the least-privilege model: By default everything is denied and then a policy is written that gives each element of the system only the access required to function. This description best describes the **strict** policy. However, such a policy is difficult to write as all system, administrator and end user interactions have to be catalogued and policies written for everything. Being the most intrusive but the most secure policy, this is a difficult but not impossible implementation in a **purpose driven LFS system** as the one we have built. Another type of policy is the **targeted** policy, which targets and confines critical system processes. Target for a policy could be critical system processes and daemons such as httpd, named, dhcpd, and mysqld. The targeted is the default policy in the Red Hat, CentOS and Fedora based distributions.

The **/etc/selinux/config** file is the main SELinux configuration file. It controls the SELinux mode and the SELinux policy to use:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of enforcing.
#       disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of these two values:
#       targeted - Targeted processes are protected,
#       mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

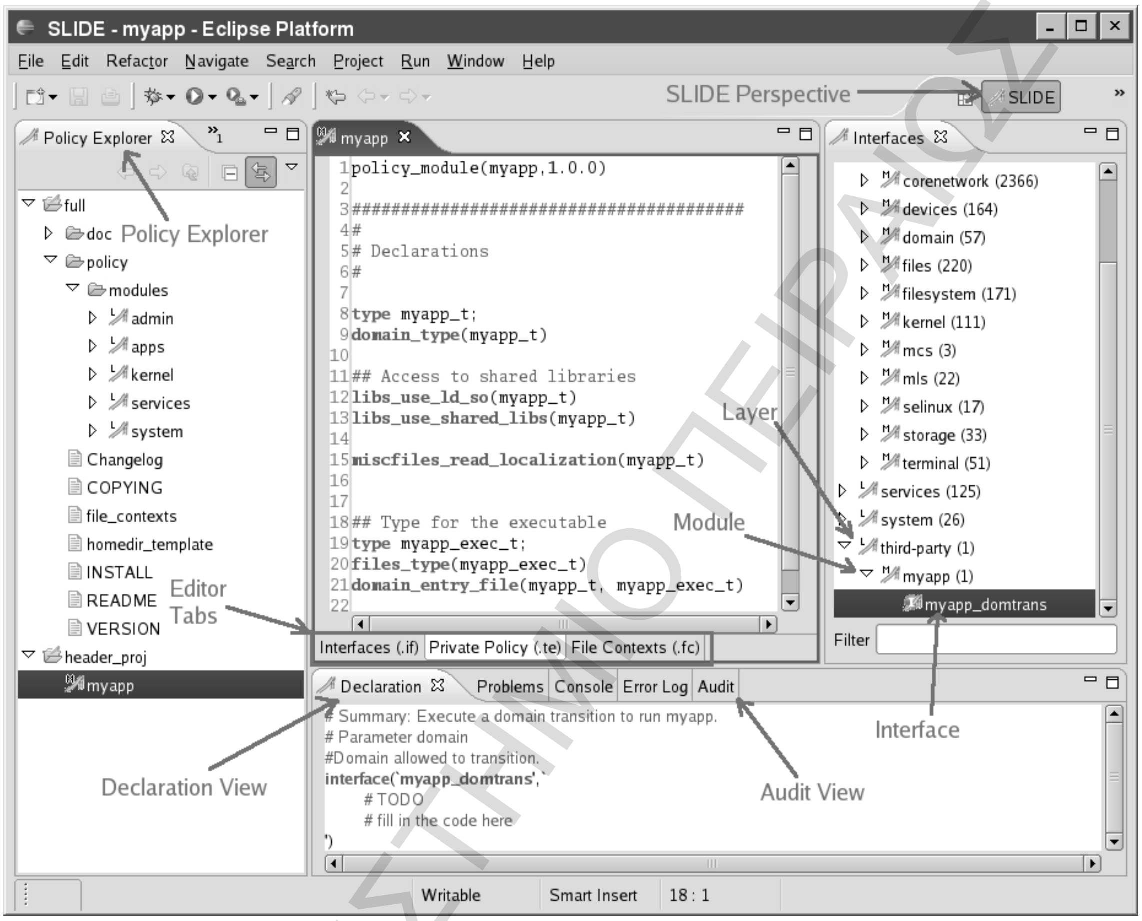
Using the SELinux userspace utilities one can configure and fine grain security policies using a c-like language structure containing classes, conditional expressions, rule lists etc. This is a rule example from a targeted policy:

```
allow user_t bin_t : file {read execute getattr};
```

Here, applications with the type user_t (subject) are allowed to read, execute, and get attributes for all objects of class file that have the type bin_t in their security context.

Creating a system specific security policy in SELinux depends on the systems function in the enterprise, its users and the enterprise itself. As our new OS has not entered such an environment, a sample SELinux policy configuration for a webserver is presented in **Appendix B**.

Writing SELinux policies in the command line can be a very frustrating process. Modern GUI tools like **SLIDE** (SELinux Policy IDE)[28] help easily develop, configure, remotely install policies and audit log monitoring.



```
policy_module(myapp,1.0.0)

########################################
#
# Declarations
#

type myapp_t;
domain_type(myapp_t)

## Access to shared libraries
libs_use_ld_so(myapp_t)
libs_use_shared_libs(myapp_t)

miscfiles_read_localization(myapp_t)

## Type for the executable
type myapp_exec_t;
files_type(myapp_exec_t)
domain_entry_file(myapp_t, myapp_exec_t)
```

```
# Summary: Execute a domain transition to run myapp.
# Parameter domain
#Domain allowed to transition.
interface(`myapp_domtrans',`
    # TODO
    # fill in the code here
')
```

Image 4.2: The SLIDE Selinux Policy Configuration Tool

Chapter 5

Other Security Enhancements

5.1 Layered Defense

A basic principle in computer security is that security has to be built in layers, combining multiple security controls to protect resources and data. No security recipe can protect a system on its own but rather as a part of larger ecosystem of measures that as a whole provide better security. In this section, additional security measures [29] that can work in parallel with SELinux to enhance the security of our system, as well as other notable security projects will be presented.

5.2 Additional Security Measures

● **Iptables**: Installing and enabling the iptables package turns on a host-based firewall on the system. While having a steep learning curve (much like SELinux), iptables is powerful due to its integration within the Kernel. It provides stateful packet inspection and network traffic filtering and has advanced logging and DoS attack prevention features. Combined with other software packets (like the squid proxy) it can even provide layer 7 (application) filtering capabilities [30].

Through the SECMARK and CONNSECMARK labeling features, iptables can work in conjunction with SELinux to apply the security policy to network entities [31] .

● **Password Policy**: Ensure a strict password policy by editing the **/etc/login.defs** file. To add security, the file must contain settings for the following:

```
PASS_MAX_DAYS 90
PASS_MIN_DAYS 6
PASS_MIN_LEN 14
PASS_WARN_AGE 7
```

● **System Logging**: Logging on Linux systems is controlled by the Syslog daemon. Logging configuration is stored in **/etc/syslog.conf**. This file identifies the level of logging and the location of the log files. Log files must be owned by the root user and group.

It is recommended that log entries are logged to a centralized log server. Centralized logging protects from deletion in the event the log files are tampered with. This is accomplished by adding the following to the syslog.conf:

```
# send to syslog server  *.emerg;*.info;*.err              @hostname<of syslog server>
```

There are several tools available to analyze and audit syslog messages. Combined with a properly configured alert system, system logging can be very helpful in identifying security threats on time.

● **SysCtl security hardening**: The /etc/sysctl.conf file provides an interface to interact with a running kernel. A lot of security enhancement can be made by editing this file, but over-tampering can lead to system crashes and should be avoided. The following lines of code can add extra networking security features:

```
net.ipv4.ip_forward = 0
net.ipv4.conf.all.send_redirects = 0
net.ipv4.conf.default.send_redirects = 0
net.ipv4.tcp_max_syn_backlog = 1280
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.conf.all.accept_source_route = 0
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.secure_redirects = 0
net.ipv4.conf.all.log_martians = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.conf.default.accept_redirects = 0
net.ipv4.conf.default.secure_redirects = 0
net.ipv4.icmp_echo_ignore_broadcasts = 1
net.ipv4.icmp_ignore_bogus_error_responses = 1
net.ipv4.tcp_syncookies = 1
net.ipv4.conf.all.rp_filter = 1
net.ipv4.conf.default.rp_filter = 1
net.ipv4.tcp_timestamps = 0
```

5.3 Notable Security Projects

● **OpenSCAP**: The Security Content Automation Protocol (SCAP) is a line of standards managed by NIST. The suite contains multiple complex data exchange formats that are to be used to transmit important vulnerability, configuration, and other security data. The Open SCAP project's goal is to provide a standardized framework of libraries and tools for maintaining the security of enterprise systems, by automatically verifying the presence of patches, system security configuration settings, and examining systems for signs of compromise [32].

● **Bastille-Linux**: The Bastille Hardening program locks down an operating system, by configuring the system for increased security and decreasing its susceptibility to compromise. It has a hardening mode in which it interactively asks the system's administrator questions, explains the topics of those questions, and builds a policy based on the user's answers. It then applies the policy to the system. In its assessment mode, it builds a report intended to teach the administrator about available security settings as well as inform the user as to which settings have been tightened. It supports all major Linux distributions [33].

Both projects are worth further exploring and testing in a suitable lab environment.

Chapter 6

Conclusions

6.1 A Security Enhanced OS

This thesis has suggested a way of constructing a secure container for sensitive government or military applications. Based on the Linux operating system, this goal was achieved in the following ways:

i. By providing the steps and proof of concept of constructing the OS from source code. This can be the basis of building dedicated, specialized systems to fulfill a specific function. The absence of any extra, never used software or useless running processes achieves a major security goal by reducing the potential exploitation surface.

ii. By presenting all modern Linux security solutions and suggesting a comparison framework based on their characteristics to choose from.

iii. By suggesting the best way to harden the security of our new OS, in way of applying the controls of the SELinux security framework and other security measures.

6.2 Further Work

Further steps have to be taken in order to consider the process of creating a security enhanced operating system complete. A more extensive lab configuration can provide the means to make the new OS bootable, apply all security enhancements and test it against known security attacks. The Open SCAP and Bastille-Linux projects mentioned in 5.3 can be evaluated.

This project can also be the basis for security enhanced devices such as mobile phones, tablets and embedded devices that are based on the Linux OS. The Security Enhancements for Android (SE for Android) project aims to identify and address critical gaps in the security of Android by enabling the use of SELinux on it [34].

References

[1] Gerard Beekmans, “Linux From Scratch: Version 7.4”, 2013, Available: www.linuxfromscratch.org

[2] PRISM (surveillance program). Available: http://en.wikipedia.org/wiki-/PRISM(surveillance_program)

[3] _NSAKEY. Available: http://en.wikipedia.org/wiki/NSAKEY

[4] BSI (German Federal Office of Information Security), “Key Requirements on Trusted Computing and Secure Boot”. Available: http://www.bmi.bund.de/SharedDocs/-Downloads/_DE/Themen/_OED_Verwaltung_/Informationsgesellschaft/-trusted_computing_eng.pdf?__blob=publicationFile

[5] FOSS A General Introduction/Linux. Available: http://en.wikibooks.org/wiki/-FOSS_A_General_Introduction/Linux

[6] The Linux Foundation, “Linux Adoption Trends: A Survey of Enterprise End Users”, 2012

[7] Michael Jang, “Security Strategies In Linux Platforms And Applications (Information Systems Security & Assurance)”, Jhones & Bartlett Learning, 2011

[8] Portable Operating System Interface (POSIX), IEEE Standards Association. Available: http://standards.ieee.org/findstds/standard/1003.1-2008.html

[9] Filesystem Hierarchy Standard (FHS), Available: http://www.pathname.com/fhs/-pub/fhs-2.3.html

[10] Linux Standard Base (LSB) Specification, Available: http://refspecs.linuxfoundation.org-/lsb.shtml

[11] LFS 7.4 Boot and Sysconfig scripts version-20130821, Available: http://www.linuxfromscratch.org/lfs/view/stable/scripts/apds01.html

[12] Dennis M. Ritchie, “On the Security of UNIX”, Bell Labs, 1979

[13] United States National Security Agency (NSA): http://www.nsa.gov/

[14] Bill McCarty, “SELinux: NSA's Open Source Security Enhanced Linux”, O’Reilly, 2004

[15] AppArmor Security Module, Available: https://help.ubuntu.com/12.04/serverguide-/apparmor.html

[16] Casey Schaufler, “The Simplified Mandatory Access Control Kernel”, Available: http://schaufler-ca.com/yahoo_site_admin/assets/docs-/SmackWhitePaper.257153003.pdf

[17] Bumjin Im, Ryan Ware, “Tizen Security Overview”, 2012, Available: http://download.tizen.org/misc/media/conference2012/tuesday/ballroom-c/2012-05-08-1600-1640-tizen_security_framework_overview.pdf

[18] TOMOYO Security Module, Available: http://tomoyo.sourceforge.jp/index.html.en

[19] YAMA Security Module, Available: https://www.kernel.org/doc/Documentation-/security/Yama.txt

[20] Brad Spengler, “The Case for grsecurity”, 2012, Available: http://grsecurity.net-/the_case_for_grsecurity.pdf

[21] Michael Fox, John Giordano, Lori Stotler, Arun Thomas , “SELinux and grsecurity: A Side-by-Side Comparison of Mandatory Access Control and Access Control List Implementations”, CS at U.Va, Available: http://www.cs.virginia.edu/~jcg8f-/SELinux%20grsecurity%20paper.pdf

[22] Erik Karlsson, “Evaluation of Linux Security Frameworks”, 2010, Available: http://www8.cs.umu.se/education/examina/Rapporter/ErikKarlsson.pdf

[23] James Morris, “Linux Kernel Security Overview”, Kernel Conference Australia, 2009, Available: http://namei.org/presentations/linux-kernel-security-kca09.pdf

[24] The SELinux Userspace Repository, Available: http://userspace.selinuxproject.org/trac

[26] Frank Mayer, Karl MacMillan, David Caplan, “SELinux by Example: Using Security Enhanced Linux”, Prentice Hall, 2006

[27] SELinux Kernel Config, Available: https://www.kernel.org/doc/menuconfig/security-selinux-Kconfig.html

[28] SELinux Policy IDE (SLIDE), Available: http://oss.tresys.com/projects/slide

[29] Red Hat, “Red Hat Enterprise Linux 6 Security Guide”, 2011, Available: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Security_Guide/Red_Hat_Enterprise_Linux-6-Security_Guide-en-US.pdf

[30] Michael Rash, “LINUX FIREWALLS, Attack Detection and Response with iptables, psad, and fwsnort”, No Starch Press, 2007

[31] SELinux Networking Support, Available: http://selinuxproject.org/page/NB_Networking

[32] The Open SCAP project, Available: http://www.open-scap.org/page/Main_Page

[33] The Bastille hardening program, Available: http://www.bastille-linux.org/

[34] The SELinux For Android Project, Available: http://selinuxproject.org/page-/SEAndroid#What_is_SE_for_Android.3F

Appendices

Appendix A

Software Packages Description

Autoconf
This package contains programs for producing shell scripts that can automatically configure source code from a developer's template. It is often needed to rebuild a package after updates to the build procedures.

Automake
This package contains programs for generating Make files from a template. It is often needed to rebuild a package after updates to the build procedures.

Bash
This package satisfies an LSB core requirement to provide a Bourne Shell interface to the system. It was chosen over other shell packages because of its common usage and extensive capabilities beyond basic shell functions.

Bc
This package provides an arbitrary precision numeric processing language. It satisfies a requirement needed when building the Linux kernel.

Binutils
This package contains a linker, an assembler, and other tools for handling object files. The programs in this package are needed to compile most of the packages in an LFS system and beyond.

Bison
This package contains the GNU version of yacc (Yet Another Compiler Compiler) needed to build several other LFS programs.

Bzip2
This package contains programs for compressing and decompressing files. It is required to decompress many LFS packages.

Check
This package contains a test harness for other programs. It is only installed in the temporary toolchain.

Coreutils
This package contains a number of essential programs for viewing and manipulating files and directories. These programs are needed for command line file management, and are necessary for the installation procedures of every package in LFS.

DejaGNU
This package contains a framework for testing other programs. It is only installed in the temporary toolchain.

Diffutils
This package contains programs that show the differences between files or directories. These programs can be used to create patches, and are also used in many packages' build procedures.

E2fsprogs
This package contains the utilities for handling the ext2, ext3 and ext4 file systems. These are the most common and thoroughly tested file systems that Linux supports.

Expect
This package contains a program for carrying out scripted dialogues with other interactive programs. It is commonly used for testing other packages. It is only installed in the temporary toolchain.

File
This package contains a utility for determining the type of a given file or files. A few packages need it to build.

Findutils
This package contains programs to find files in a file system. It is used in many packages' build scripts.

Flex
This package contains a utility for generating programs that recognize patterns in text. It is the GNU version of the lex (lexical analyzer) program. It is required to build several LFS packages.

Gawk
This package contains programs for manipulating text files. It is the GNU version of awk (Aho-Weinberg-Kernighan). It is used in many other packages' build scripts.

Gcc
This package is the Gnu Compiler Collection. It contains the C and C++ compilers as well as several others not built by LFS.

GDBM

This package contains the GNU Database Manager library. It is used by one other LFS package, Man-DB.

Gettext
This package contains utilities and libraries for internationalization and localization of numerous packages.

Glibc
This package contains the main C library. Linux programs would not run without it.

GMP
This package contains math libraries that provide useful functions for arbitrary precision arithmetic. It is required to build Gcc.

Grep
This package contains programs for searching through files. These programs are used by most packages' build scripts.

Groff
This package contains programs for processing and formatting text. One important function of these programs is to format man pages.

GRUB
This package is the Grand Unified Boot Loader. It is one of several boot loaders available, but is the most flexible.

Gzip
This package contains programs for compressing and decompressing files. It is needed to decompress many packages in LFS and beyond.

Iana-etc
This package provides data for network services and protocols. It is needed to enable proper networking capabilities.

Inetutils
This package contains programs for basic network administration.

IProute2
This package contains programs for basic and advanced IPv4 and IPv6 networking. It was chosen over the other common network tools package (net-tools) for its IPv6 capabilities.

Kbd
This package contains key-table files, keyboard utilities for non-US keyboards, and a number of console fonts.

Kmod
This package contains programs needed to administer Linux kernel modules.

Less
This package contains a very nice text file viewer that allows scrolling up or down when viewing a file. It is also used by Man-DB for viewing manpages.

Libpipeline
The Libpipeline package contains a library for manipulating pipelines of subprocesses in a flexible and convenient way. It is required by the Man-DB package.

Libtool
This package contains the GNU generic library support script. It wraps the complexity of using shared libraries in a consistent, portable interface. It is needed by the test suites in other LFS packages.

Linux Kernel
This package is the Operating System. It is the Linux in the GNU/Linux environment.

M4
This package contains a general text macro processor useful as a build tool for other programs.

Make
This package contains a program for directing the building of packages. It is required by almost every package in LFS.

Man-DB
This package contains programs for finding and viewing man pages. It was chosen instead of the man package due to superior internationalization capabilities. It supplies the man program.

Man-pages
This package contains the actual contents of the basic Linux man pages.

MPC
This package contains functions for the arithmetic of complex numbers. It is required by Gcc.

MPFR
This package contains functions for multiple precision arithmetic. It is required by Gcc.

Ncurses
This package contains libraries for terminal-independent handling of character screens. It is often used to provide cursor control for a menuing system. It is needed by a number of packages in LFS.

Patch
This package contains a program for modifying or creating files by applying a patch file typically created by the diff program. It is needed by the build procedure for several LFS packages.

Perl
This package is an interpreter for the runtime language PERL. It is needed for the installation and test suites of several LFS packages.

Pkg-config
This package provides a program to return meta-data about an installed library or package.

Procps-NG
This package contains programs for monitoring processes. These programs are useful for system administration, and are also used by the LFS Bootscripts.

Psmisc
This package contains programs for displaying information about running processes. These programs are useful for system administration.

Readline
This package is a set of libraries that offers command-line editing and history capabilities. It is used by Bash.

Sed
This package allows editing of text without opening it in a text editor. It is also needed by most LFS packages' configure scripts.

Shadow
This package contains programs for handling passwords in a secure way.

Sysklogd
This package contains programs for logging system messages, such as those given by the kernel or daemon processes when unusual events occur.

Sysvinit
This package provides the init program, which is the parent of all other processes on the Linux system.

Tar
This package provides archiving and extraction capabilities of virtually all packages used in LFS.

Tcl
This package contains the Tool Command Language used in many test suites in LFS packages. It is only installed in the temporary toolchain.

Texinfo
This package contains programs for reading, writing, and converting info pages. It is used in the installation procedures of many LFS packages.

Udev
This package contains programs for dynamic creation of device nodes. It is an alternative to creating thousands of static devices in the /dev directory.

Util-linux
This package contains miscellaneous utility programs. Among them are utilities for handling file systems, consoles, partitions, and messages.

Vim
This package contains an editor. It was chosen because of its compatibility with the classic vi editor and its huge number of powerful capabilities. An editor is a very personal choice for many users and any other editor could be substituted if desired.

XZ Utils
This package contains programs for compressing and decompressing files. It provides the highest compression generally available and is useful for decompressing packages XZ or LZMA format.

Zlib
This package contains compression and decompression routines used by some programs.

Appendix B

A Sample SELinux Policy

This example shows the SELinux policy written for a webserver (InterNetNews server)[1]. After the policy is saved, we run "make load" to apply it.

/etc/selinux/domains/program/innd.te:

```
#DESC INN - InterNetNews server
#
# Author:  Faye Coker <faye@lurking-grue.org>
# X-Debian-Packages: inn
#
################################

# Types for the server port and news spool.
#
type innd_port_t, port_type;
type news_spool_t, file_type, sysadmfile;


# need privmail attribute so innd can access system_mail_t
daemon_domain(innd, `, privmail')

# allow innd to create files and directories of type news_spool_t
create_dir_file(innd_t, news_spool_t)

# allow user domains to read files and directories these types
r_dir_file(userdomain, { news_spool_t innd_var_lib_t innd_etc_t })

can_exec(initrc_t, innd_etc_t)
can_exec(innd_t, { innd_exec_t bin_t })
ifdef(`hostname.te', `
can_exec(innd_t, hostname_exec_t)
')

allow innd_t var_spool_t:dir { getattr search };

can_network(innd_t)

can_unix_send( { innd_t sysadm_t }, { innd_t sysadm_t } )
allow innd_t self:unix_dgram_socket create_socket_perms;
allow innd_t self:unix_stream_socket create_stream_socket_perms;
can_unix_connect(innd_t, self)

allow innd_t self:fifo_file rw_file_perms;
allow innd_t innd_port_t:tcp_socket name_bind;

allow innd_t self:capability { dac_override kill setgid setuid
net_bind_service };
allow innd_t self:process setsched;
allow innd_t { bin_t sbin_t }:dir search;
allow innd_t usr_t:lnk_file read;
allow innd_t usr_t:file { getattr read ioctl };
allow innd_t lib_t:file ioctl;
allow innd_t { etc_t resolv_conf_t }:file { getattr read };
allow innd_t { proc_t etc_runtime_t }:file { getattr read };
allow innd_t urandom_device_t:chr_file read;

allow innd_t innd_var_run_t:sock_file create_file_perms;

# allow innd to read directories of type innd_etc_t (/etc/news/(/.*)?
and symbolic links with that type
etcdir_domain(innd)

# allow innd to create files under /var/log of type innd_log_t and
have a directory for its own files that
# it can write to
logdir_domain(innd)

# allow innd read-write directory permissions to /var/lib/news.
var_lib_domain(innd)

ifdef(`crond.te', `
system_crond_entry(innd_exec_t, innd_t)
')
```

/etc/selinux/file_contexts/program/innd.fc

```
/usr/sbin/innd.*        --      system_u:object_r:innd_exec_t
/var/run/innd(/.*)?             system_u:object_r:innd_var_run_t
/etc/news(/.*)?                 system_u:object_r:innd_etc_t
/etc/news/boot          --      system_u:object_r:innd_exec_t
/var/spool/news(/.*)?           system_u:object_r:news_spool_t
/var/log/news(/.*)?             system_u:object_r:innd_log_t
/var/lib/news(/.*)?             system_u:object_r:innd_var_lib_t
/usr/sbin/in.nnrpd      --      system_u:object_r:innd_exec_t
/usr/lib/news/bin/.*    --      system_u:object_r:innd_exec_t
/usr/bin/inews          --      system_u:object_r:innd_exec_t
/usr/bin/rnews          --
system_u:object_r:innd_exec_t_r:innd_exec_t /usr/bin/rnews
--      system_u:object_r:innd_exec_t

make -C file_contexts/file_contexts
```

/net_contexts

```
ifdef(`innd.te', `portcon tcp 119 system_u:object_r:innd_port_t')
```

[1] Faye Coker, The Guide to Writing SELinux Policy, 2004, Available: http://www.linuxtopia.org-/online_books/writing_SELinux_policy_guide/index.html