SE Linux Policy Writing

Russell Coker <russell@coker.com.au>
Internet and Security Consulting

I prefer an interactive format for my presentations, please ask questions at any time.

Policy Overview

- Policy defines types for file system objects, domains for processes, roles to limit the domains which can be entered, and identities to limit the roles which can be used
- Policy defines rules for what access each domain has to each type. By default nothing is allowed and every attempt to perform a denied operation is logged. Policy can allow operations, and can specify that some denied operations are not to be audited or that some permitted operations are to be audited.
- The policy is loaded into the kernel at boot time by /sbin/init and a new policy may be loaded by the administrator at any time
- The complete operation of the system is determined by the policy and the type labeling of the file systems

Policy Size and Compilation

- Policy is at the core of SE Linux. The Fedora Core 5 targeted policy has about 52,000 allow rules, a minimal policy may have as little as 30,000 rules.
- For the Strict policy there needs to be a policy for each daemon and for each significant program that a user may run from a login session. This gives greater restriction on access and consequently a greater number of rules. The default for FC5 is about 100,000 allow rules.
- Policy is written at a high level with M4 macros. In FC5 the source of the policy (strict, targeted, and MLS) is about 40,000 lines.
- The result of the M4 processing is either a single file that is compiled with "checkpolicy" into a binary form that can be loaded into the kernel or a set of files that are compiled with "checkmodule" for the modular policy.

Policy Binary

- One allow or dontaudit line in the policy source can result in many rules (attributes)
- Multiple allow and dontaudit lines can be combined into a single rule if they all refer to the same contexts and class
- The checkpolicy/checkmodule program determines the most efficient way of converting complex rules in the policy source to simple rules that the kernel can process
- Policy modules have dependencies and interfaces

Policy Aims

- Strict policy gives minimal privs to every daemon (a separate domain for each daemon) and creates separate user domains for programs that are important to system security such as GPG, X, ssh, etc

- Domains for user programs can be for untrusted programs (EG UML virtual servers and games), or for programs that deal with secret data (such as GPG)

- Targetted policy locks down a small number of daemons that are significant for security but leaves most programs unrestricted. All programs run in the user session are unrestricted.

Policy Syntax 1/5

- Define an attribute that applies to 0 or more types:
  - attribute ATTRIB;
- The most common attributes are domain (for processes), file_type (for file system objects), sysadmfile (for files the administrator may write to), port_type (for labeling network ports), device_type (for device nodes), and userdomain (which applies to all domains that may be used for a login session)
- Define a type for a process/port/file:
  - type NAME [, ATTRIB [, ATTRIB ...]];
  - type PROCESS_t, domain;
  - type SERVICE_PORT_t, port_type;
  - type A_FILE_t, file_type, sysadmfile;
- There are currently only two type declarations in the Fedora policy which don't have attributes, they are proc_kmsg_t and proc_kcore_t (for /proc/kmsg and /proc/kcore). It is extremely unlikely that you will have a good reason to define types without attributes.

Policy Syntax 2/5

- To allow an access use the following:
  - allow DOMAIN TYPE:CLASS OPERATION;
  - allow mount_t file_type:dir search;
  - allow mount_t file_t:file { getattr read unlink };
- The DOMAIN may be the name of a domain, an attribute representing 0 or more domains, or a list of domains and/or attributes. Rules that apply to an attribute that represent 0 domains are OK, they are silently discarded at compile time. A List is enclosed in curly brackets '{' and '}', brackets within brackets are legal and work in the way you would expect.
- The TYPE is the type of a file system object or domain of a process that is being acted upon. Lists and attributes work in the same manner as for the DOMAIN.
- The CLASS describes the category of the object that is being referred to, the most common classes that you will use when writing policy are the file-system classes, they are file, dir, lnk_file, chr_file, blk_file, sock_file, and fifo_file.

Policy Syntax 3/5

- It is often the case that an operation will be permitted for a given type in one class but not for another object of the same type but a different class (EG allowing writes to a file but not writes to a directory of the same type). For most of the file-system classes you can guess what they mean and how to use them. The only non-obvious use is that fifo_file applies to pipes created with the pipe() system call as well as to fifo's on disk.
- A role is declared implicitly when domains are assigned to it with the following syntax:
  - role ROLE-NAME types TYPE-LIST;
- Where ROLE-NAME is the name of the role and TYPE-LIST is the list of domains that are permitted in the role. For example:
  - role system_r types syslogd_t;
  - role user_r types user_t;

Policy Syntax 4/5

- The identity is defined in the users file with the following syntax:
  - user USER-NAME roles ROLE-LIST;
- Where USER-NAME is the name of the user and ROLE-LIST is the list of roles permitted for the identity. For example:
  - user jdoe roles { user_r };
  - user jadmin roles { staff_r sysadm_r system_r };
- Identity declarations should list the default role first as that determines the type of the home directory
- NB Only specify identities in the policy for FC4 and RHEL4, for FC5 and RHEL5 use semanage

Policy Syntax 5/5

In the reference policy interfaces allow modules to depend on other modules or have optional policy that is only active when the other module is active. From radius.if:

```
interface(‘radius_use’,‘
  gen_require(‘type radiusd_t;’)
  allow $1 radiusd_t:udp_socket sendto;
  allow radiusd_t $1:udp_socket recvfrom;
  allow radiusd_t $1:udp_socket sendto;
  allow $1 radiusd_t:udp_socket recvfrom;
’)
```

From portslave.te:

```
optional_policy(‘radius_use(portslave_t)’)
```

Dir/File Macros 1/2

- For file-system classes the permitted operations are ioctl, read, write, create, getattr, setattr, lock, relabelfrom, relabelto, append, unlink, link, rename, execute, swapon, quotaon, and mounton. All of these are permitted to apply to all file-system classes in the policy compilation, although some combinations (such as swapon for dir class) do not make sense. The dir class has the additional accesses of add_name, remove_name, reparent, search, and rmdir.
- For most policy you won't use the operations directly, you will use a macro that expands to a list of operations that match your needs. Here are some of the macros defined for file access:
  - r_file_perms { read getattr lock ioctl }
  - rx_file_perms { read getattr lock execute ioctl }
  - rw_file_perms { ioctl read getattr lock write append }
  - ra_file_perms { ioctl read getattr lock append }
  - create_file_perms { create ioctl read getattr lock write setattr append link unlink rename }

Dir/File Macros 2/2

Here are the macros for directory access:

- r_dir_perms { read getattr lock search ioctl }
- rw_dir_perms { read getattr lock search ioctl add_name remove_name write }
- ra_dir_perms { read getattr lock search ioctl add_name write }
- create_dir_perms { create read getattr lock setattr ioctl link unlink rename search add_name remove_name reparent write rmdir }

Networking

- To grant network access the macros can_network_server_tcp() can_network_client_tcp() can_network_tcp() can_network_udp() can_network_server()and can_network_client() are used, they take a single parameter which is the name of the domain (or an attribute or list of domains) which should be granted network access. This allows basic TCP/IP networking via the tcp_socket and udp_socket classes.

- The two most common network classes are unix_stream_socket and unix_dgram_socket which are used for Unix domain sockets (commonly used within a process by library calls)

Well Known Ports

- When a daemon listens to a port you will create a type for the port and in the net_contexts file put a line of the form:
  - portcon PROTOCOL NUMBER system_u:object_r:PORT_TYPE
- Where PROTOCOL is either tcp or udp, the NUMBER is the port number, PORT_TYPE is the type to be assigned to the port.

- To allow listening on the port use a line of the form:
  - allow DOMAIN PORT_TYPE:PROTOCOL name_bind;
  - EG: allow slapd_t ldap_port_t:tcp_socket name_bind;

- Every well-known port that a daemon listens on should have a type assigned to it to prevent other daemons from binding to it. Otherwise the wrong daemon can bind to it maliciously (if controlled by an attacker) or accidentally (through binding to an unspecified port number).

Assertions

- The policy language allows very powerful rules that if misunderstood could allow more access than is desired. To prevent such mistakes there are assertion rules of the following form:
  - neverallow DOMAIN TYPE:CLASS OPERATION;
- If there is an allow statement that conflicts with a neverallow statement then the policy will not compile.
- A common feature in an assertion is the logical not operator represented by the '~' character, for example "~{ domain unlabeled_t }" means every type that is not a domain and not unlabeled.
- Another common feature in assertions is a logical subtraction which means all the types in one group minus any matching types which may be in a second group. The following assertion prevents domains that don't have the auth_write attribute from writing to /etc/shadow:
  - neverallow { domain -auth_write } shadow_t:file ~r_file_perms;

Auditing

- By default every operation that is denied by SE Linux is audited and every operation that is permitted is not
- To audit an allowed operation use the following syntax. NB an auditallow rule does not imply an allow rule:
  - allow DOMAIN TYPE:CLASS OPERATION;
  - auditallow DOMAIN TYPE:CLASS OPERATION;
- To avoid auditing an operation that is not permitted (usually because it happens very frequently) do the following:
  - dontaudit DOMAIN TYPE:CLASS OPERATION;
- Note that if Unix permissions do not permit an operation then the SE Linux kernel code will never see it, and therefore it will not be audited.

Important Domains/Types

- kernel_t used for the kernel and for init before it loads the policy and re-exec’s itself
- init_t used for init after it loads the policy and re-exec’s itself
- unlabeled_t for processes and files who’s context has become invalid due to a policy change
- file_t for files that have never been labeled

Important Attributes 1/2

- domain is assigned to all types for processes apart from unlabeled_t, used as the target in some allow rules and in assertions
- privlog is assigned to all domains that can use syslog, grants permission to write to /dev/log
- auth grants read access to /etc/shadow - generally programs should not require this, any program that uses PAM should use auth_chkpwd, any program that doesn't use PAM should ideally be modified to use unix_chkpwd directly
- auth_write grants read/write access to /etc/shadow
- auth_chkpwd allows a system domain (domain running in the system_r role) to authenticate users against /etc/shadow by running unix_chkpwd (which runs in a domain that is permitted to read shadow_t
- etc_writer is assigned to all domains that may write files of type etc_t
- admin identifies every administrator domain
- file_type identifies all types assigned to files, directories, links, sockets, and fifo's on disk

Important Attributes 2/2

- device_type identifies all types assigned to device nodes
- sysadmfile identifies all types that the administrator may have full access to, such access is granted automatically for every administrator domain (of which there is only one at the moment)
- userdomain identifies every user domain used for a login session
- unpriv_userdomain identifies every user domain apart from sysadm_t probably not much use since the subtraction operator was introduced into the policy language
- privmodule allows a system domain to load modules with modprobe/insmod
- privhome allows a domain to have access to the base types for all user home directories, generally only the administrator and mail servers need this

Mail Servers 1/2

- A mail server interacts with most programs on the system. To alleviate the risk of local DOS the program that receives mail from a user runs in it's own domain.
- Separate domain for receiving local mail from each user role, such domains can not communicate with each other
- Mail server also interacts with the rest of the system for local delivery, SMTP delivery, and SMTP reception

Mail Servers 2/2

- Mail related attributes:
  - user_mail_domain for user_mail_t, staff_mail_t, sysadm_mail_t, etc - all the domains used to read the user's files to send to the local mail server
  - mta_user_agent for domains that are part of an MTA which need to read user files for local mail receipt - executed from user_mail_domain
  - mta_delivery_agent for MTA domains that deliver mail to the user's home directory
  - mail_server_sender for the domain in the MTA which makes outbound port 25 connections - for anti-virus etc
  - mail_server_domain for MTA domains that listen on port 25

User Domains

- The macro full_user_role() creates all domains and types for a user and a role to match. The default policy has full_user_role(user) to create the role user_r, the domain user_t, the types user_home_dir_t, user_home_t, and user_tmp_t as well as many others.
- To add a new user role named professor you need full_user_role(professor) to create all necessary domains and types and you need to modify the macro in_user_role() to cause the domains for passwd_t, newrole_t and other critical system programs to be in the professor_r role.

User Domains for Applications

- Sometimes it is required that an application which is run by a user have a different security context. For example GPG runs in a different domain so that the main user domain can't access the secret key (to make it more difficult to steal the key).
- The file domains/program/gpg.te has the following policy:
  - type gpg_exec_t, file_type, sysadmfile, exec_type;
- The file file_contexts/program/gpg.fc has the following type labels:
  - HOME_DIR/\.gnupg(/.+)?   system_u:object_r:ROLE_gpg_secret_t
  - /usr/bin/gpg   system_u:object_r:gpg_exec_t
- The file macros/base_user_macros.te has the following:
  - ifdef('gpg.te', 'gpg_domain($1)')
- If the file gpg.te is included in the policy build then the macro 'gpg.teâ will be defined, in which case we want to call the macro gpg_domains() and pass the name of the user role as the first parameter (referred to as $1 in the M4 macro language).

Defining a Macro for a Domain definition

- I use the macro gpg_domain() as an example. I have included only the most relevant policy, if you read the policy source you will notice other things I have omitted for this example.

- The file macros/program/gpg_macros.te has an M4 macro definition of the form
  - define(‘gpg_domain’, ‘
  - some policy...
  - ’)

- The next slide has the policy in the macro.

Macro Policy for a Domain definition 1/2

- Declare the domain and the type for the files it uses and permit the domain in the role.
  - type $1_gpg_t, domain, privlog;
  - type $1_gpg_secret_t, file_type, homedirfile, sysadmfile;
  - role $1_r types $1_gpg_t;
- Transition to user_gpg_t from user_t when running gpg_exec_t.
  - domain_auto_trans($1_t, gpg_exec_t, $1_gpg_t)
- Allow the user to see their gpg process in ps output and kill it.
  - allow { $1_gpg_t $1_t } $1_gpg_t:process signal;
  - can_ps($1_t, $1_gpg_t)
- Allow the domain to use it's terminal device:
  - allow $1_gpg_t { $1_devpts_t $1_tty_device_t }:chr_file rw_file_perms;
  - allow $1_gpg_t privfd:fd use;

Macro Policy for a Domain definition 2/2

- Allow gpg to use shared objects (libc6, libz, and libbz2), and allow it to download public keys from the Internet.
  - uses_shlib($1_gpg_t)

    can_network($1_gpg_t)
- When user_gpg_t creates ~/.gnupg have it labelled as user_gpg_secret_t.
  - file_type_auto_trans($1_gpg_t, $1_home_dir_t, $1_gpg_secret_t, dir)
- Give user_gpg_t read/write access to user_gpg_secret_t directories and allow it to create files of the same type.
  - rw_dir_create_file($1_gpg_t, $1_gpg_secret_t)
- Allow user_gpg_t to read files created by user_t and write output files that user_t can read.
  - file_type_auto_trans($1_gpg_t, $1_home_dir_t, $1_home_t, file)
  - file_type_auto_trans($1_gpg_t, tmp_t, $1_tmp_t, file)

AVC Messages

- When an access is not permitted by SE Linux an audit message is logged, here is a sample from running the strict policy:
  - audit(1089889979.989:0): avc: denied { write } for pid=10317 exe=/usr/bin/vim name=etc dev=hda1 ino=162881 scontext=root:user_r:user_t tcontext=system_u:object_r:etc_t tclass=dir
- This tells us that at time 1089889979.989 seconds since 1970-01-01 the process vim which had pid 10317 tried to write to an object named etc (which had the Inode number 162881) of class dir on the file system hda1. Vim had the context root:user_r:user_t which means that someone logged in as root with an unprivileged user role. The context of the directory was system_u:object_r:etc_t.
- The program audit2allow can be used to convert AVC messages to allow rules, the above message would be converted to:
  - allow user_t etc_t:dir write;
- But don't do this. Most times the output of audit2allow is not suitable for adding to your policy, it's just an indication of what's being attempted.

Policy Analysis

- apol from Tresys http://www.tresys.com/ is the most functional tool for analyzing policy. It allows reading a binary policy or a policy.conf file and analyzing it.

- Analyzing a policy source file is more useful because it can tell you which line of source permits an action, and show information on attributes. When the policy is compiled such data is discarded.

- Supported operations include discovering which rule permits an operation (which is extremely difficult to do in any other way when attributes are used), discovering whether an operation is permitted, and data flow analysis.

- slat from MITRE is a tool to analyse data flows in SE Linux policy, currently it works on the policy.conf file but will probably be changed to work on policy binaries. It produces an analysis of the policy which then needs to be processed by another tool.

Work to be Done

- Policy is at the core of SE Linux, any new developments in the kernel or applications need matching policy changes
- New features in daemons and new interactions between daemons requires new policy, incremental change to cope with new versions is constant
- Policy has to be audited to make sure that it meets it’s goals. Tresys has developed a tool to analyse policy to help achieve this goal.
- Adding support for Security Enhanced X
- Writing more policy for MLS
- Policy to give similar functionality to BSD secure levels

End of Part 3 - Q/A

- #selinux on irc.freenode.net
- http://www.nsa.gov/selinux/ Official SE Linux web site
- http://www.coker.com.au/selinux/ My SE Linux web pages

- Russell Coker <russell@coker.com.au>