Breaking the Ice with SELinux

Eli Billauer

December 8th, 2008

Introduction

What SELinux is

- In a nutshell: A machine that tells you permission is denied.
- Implementation: A kernel module + (a lot of) supporting utilities + (a lot of) configuration files
- The kernel module is asked for permissions before certain operations are about to happen ("hooks")
- Fine-grained
- SELinux doesn't care about classic user names and groups

The goals of this lecture

- Make the existing docs understandable
- Explain the basics of writing rules
- Show how to play around with SELinux without compromising the system's security
- Demonstrate a quick method for limiting an application's permissions to minimum, by making an SELinux module

Why SELinux is good

- Resolution: Give the application permissions as necessary, no more
- Targeting: Let everyone do whatever they want, except for a few applications with exploit potential
- Jailing: The application is not likely to escape from its state of limited permissions
- Flexibility: The machine can be configured for other purposes, such as controlling information access for employers
- Alert: The administrator can catch unexpected behavior at early stages of an attack (adversary "looking around").

Problems with SELinux

- Complicated
- Unhelpful documentation (that's an understatement)
- ... and therefore very difficult to learn
- Careless hacking can create huge security holes
- May cripple applications without the user understanding why
- Is brought to end users with a "trust us, we're the experts"
- ... and leaves very little choice unless you want to dive in

SELinux: What's in the package

- The kernel security core: The LSM (Linux Security Modules)
- "The example policy": The basic security rules used
- Policy modules: Rules specific to certain applications
- Filesystem extension to allow extra attributes (the context) for each file.
- User-space utilities and daemons directly interacting with the LSM.
- Housekeeping utilities (essential to configure SELinux, but don't interact with the kernel, such as the rules compiler).
- SELinux aware versions of common utilities: ls, ps, id, find, etc.

Do I have SELinux?

- If you have a `/selinux` directory with something in it, SELinux is loaded in the kernel.
- Also try the `sestatus` command. This is what you get on Fedora Core 9 by default:

```
[eli@rouge home]$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 22
Policy from config file:        targeted
```

Note that SELinux is enabled and enforcing. Simply put, we're on.

Concepts

The Policy

- Policy – A set of declarations and rules, telling the SELinux core in the kernel what is permitted and how to behave in different situations
- Targeted policy – A policy based upon the paradigm, that only a few selected applications should be restricted by SELinux. All other activity relies on good old UNIX security
- Strict policy – A policy which attempts to control all activity with SELinux

The commonplace (and sane?) policy is a Targeted policy.

How the policy is consumed

- The policy is compiled in user space
- The m4 macro preprocessor is used prior to compilation (optional)
- The initial policy binary is loaded by `init` at boot
- Policy modules (binaries) can be loaded and unloaded at any time

The Context

- SELinux marks every process, file, pipe, socket, etc. with a piece of information called the *context*.
- SELinux allows or denies actions based upon rules saying "a process of context X can do so and so in relation with something with context Y"
- The context is completely *unrelated* to classic UNIX user ID, group ID or whatever.
- In particular: su, sudo and suid-bit games don't change the context. To SELinux you remain who you were before.
- In short: In SELinux, the context is everything.

The Context (cont.)

- The context consists of three parts: The user, the role, and the type
- In a commonplace policy, 99% of the decisions are made based upon type *only*
- When the context applies to a process, the type is called "the *domain*"
- There is no practical difference between a type and a domain
- All three components are just names. The policy rules gives them significance.
- In particular, if an object has the same type as a process' domain, this means something only if the policy explicitly says so (it usually does).
- All users, roles and types can be applied to any object (given the permissions), since they are just names

A simple session

On a SELinux-enabled system:

```
[eli@rouge home]$ ls -Z
drwxrwxr-x  eli eli unconfined_u:object_r:user_home_t:s0 mydir
-rw-rw-r--  eli eli unconfined_u:object_r:user_home_t:s0 myfile

[eli@rouge home]$ ls -Z /etc/passwd
-rw-r--r--  root root system_u:object_r:etc_t:s0       /etc/passwd

[eli@rouge home]$ ps -Z
LABEL                             PID TTY          TIME CMD
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 22599 pts/9 00:00:00 bash
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 22623 pts/9 00:00:00 ps

[eli@rouge home]$ id
uid=1010(eli) gid=500(eli) groups=500(eli) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

The nuts and bolts

What SELinux is actually doing

Loaded in the kernel, the Linux Security Module performs three ongoing tasks, based upon the rules loaded from user space (i.e. the Policy):

- Grant or deny access permission to processes requesting to perform action on objects
- Grant or deny permission for context changes of objects and processes.
- Decide what context to give to new objects and processes at their creation.

SELinux permissions are given on top of classic UNIX permissions. An action will take place only if both permissions are granted.

Enforcing vs. permissive mode

- Enforcing mode – The kernel refuses any action for which SELinux denies permission
- Permissive mode – SELinux only writes denial log messages, but the kernel ignores its denials (only classic UNIX permissions take effect)
- By default, any sane system will boot in enforcing mode
- As root, switch to permissive mode with `setenforce 0`
- ... and back to enforcing with `setenforce 1`
- Booting the system in permissive mode: Use the `enforcing=0` kernel boot parameter

The foodchain: Roles, users and types

- SELinux policy constrains which (SELinux) users can get which roles
- It's common but not necessary, that each SELinux user can and will have one single role
- The role limits which domains (types) its owner can enter
- RBAC (Role-Based Access Control): Restrict user's permissions by allocating roles, which in turn limit their variety of types, and hence limit their actions.
- The commonplace Linux policy is Type Enforced (TE), so roles and users are of little importance.

The foodchain: Roles, users and types (cont.)

- Upon login (not su), the shell process is given a SELinux user and a role, typically unconfined_u and unconfined_r.
- These are most likely to remain throughout the session for all child processes.
- Processes created by init or crond are likely to get system_u and system_r

So why should we care about users and roles at all?

- When declaring a new type, we must explicitly allow them to the relevant roles. More about this later.
- `seinfo -r` will print out all roles known to the system
- Again: Remember that the login user and SELinux user are unrelated, unless otherwise configured.
- Roles and user are currently meaningless on objects (files, sockets etc.)
- The only current rule says that except for privileged domains, the user of an object can't be changed (see the "constraints" file in the policy source tree).
- Bottom line: Let's keep our eyes on the types

SELinux objects and classes

- The term “object” in SELinux stands for files, directories, file descriptors, pipes, sockets, network interfaces and many more.
- An object is the thing some process asks for permission to do something on
- There are more than 70 *classes* of SELinux objects
- Each class defines which permissions are applicable
- There is a “process” class, but in the jargon, a process is usually not considered an object
- ... but rather the *subject* (as in English grammar terminology)
- Think subject, action, object as in “The cat drinks the milk”
- This confusion does not affect the policy rules’ syntax

Multi Level/Category Security

```
[eli@rouge home]$ ls -Z
drwxrwxr-x  eli eli unconfined_u:object_r:user_home_t:s0 mydir
-rw-rw-r--  eli eli unconfined_u:object_r:user_home_t:s0 myfile

[eli@rouge home]$ cat /selinux/mls
1
```

- So I have MLS on!
- MLS and MCS is the forth element in the context (s0 in the example above).
- These mechanisms are intended to prevent users from leaking information by mistake (think "top secret" stamp)
- For example, the mail application may be prevented to read sensitive files
- Can be ignored if we don't use it (so we shall)
- Implemented with "mlsconstraint" rules in mls and mcs files in the policy source directory

How SELinux decides what to permit

The SELinux kernel module will permit an operation if and only if:

1. A permission rule (`allow` or `allowaudit`) matches the types and classes of the involved elements.
2. None of the contraint rules is violated

Remarks:

- The decisions are cached in the Access Vector Cache
- As of today's targeted policy, the constraints are very basic, meaning that only the types carry a significance

How files get their context

- The context is stored for each file as attributes on an extended filesystem, XFS (man attr)
- As a starting point, the `setfiles` utility sets the context to all files, according to some configuration file (typically /etc/selinux/targeted/contexts/files/file_contexts)
- This is called *relabeling*
- Don't edit this file directly. Instead, use `semanage fcontext` to permanently change the context of files and directories (regular expression)
- Installing a policy module may also alter file contexts permanently
- `restorecon` does the same as `setfiles`, but is intended for a few files only (mostly to fix small mismatches)

How files get their context (cont.)

- Use chcon to alter some file's context without changing the configuration files. Note that this change is temporary until the next relabeling.
- The policy includes rules which determine file types at creation (more about this later)
- **Contradictions between policy rules and relabeling configuration files are possible and dangerous.**
- Filesystems which can't carry extended attributes get a uniform context, depending on options of the mount operation and system configuration files (e.g. VFAT, NFS, Samba, ISO)
- Note that tar doesn't store and extract contexts unless explicit flags are given

Policy syntax

The “allow” rule

```
allow Source Target:Class Permission;
```

- This means “grant Permission to a process of domain (type) Source on objects of type Target and class Class”

Example:

```
allow unconfined_t mytype_t:file read ;
```

- ... which means “allow processes in domain (type) unconfined_t read permission on files of type mytype_t”
- There is no need to write permission rules from scratch
- `audit2allow` will do most of the work for us
- It’s extremely important to understand what the rules say

Other allow-likes

- `auditallow` – Exactly like `allow`, but makes entries in the log (as in denials)
- `dontaudit` – This will not grant permission, but not log anything either
- `neverallow` – Not really a rule, but tells the rule compiler to exit with an error, if the specified permissions are granted by other rules. Used as an extra safeguard against bugs in the policy
- Except for the opening keyword, the three above have the same syntax as `allow`
- In case of contradiction between rules, the rule appearing *later* takes effect.
- **Not clear whether these work with modules**

Type transitions for objects

- Every created object has a default context
- For example, files and directories are created by default with their parent directory's context
- It's often desireable that the type of the new object will depend on who created it. "Who" means what domain (type) the process had.
- For example: If the X server creates a file in the `/tmp` directory, it should have type `xdm_tmp_t`, but if a "normal user" process does so, it should be `user_tmp_t`
- The solution: Type transitions

Type transitions for objects (cont)

`type_transition Source Target:Class new_type;`

- This means “any object of class Class, which is created by a process in the domain (type) Source, and would by default get the type Target, will get the type new_type instead”

Example:

`type_transition sshd_t tmp_t:file sshd_tmp_t;`

- ... which means that if a process running in the `sshd_t` domain (most likely the ssh deamon) creates a plain regular file which should have gotten the `tmp_t` type (most likely because it’s in the `/tmp` directory), it should get the `sshd_tmp_t` instead.
- Note that this is not a permission rule. Rather, this tells SELinux itself to perform an action.

Type transitions for objects (cont)

- The type transition for objects doesn't require an additional permission rule
- But several other actions need permission:
- Read-write access to the parent directory
- Creating a new file or directory with the new type
- To make things easier, a macro bundles the type transition statement with the permissions, `file_type_auto_trans`
- Paraphrasing the last example, the following macro statement covers a variety of file types (plain files, directories, symlinks etc) and also handles the permissions. All in one:

```
file_type_auto_trans(sshd_t, tmp_t, sshd_tmp_t);
```

Domain transitions for processes

```
type_transition Source Target:process new_type;
```

- This means "when a process in the domain Source executes a file of type Target, change the process' domain to new_type.
- Occurs when an application is executed – an `exec()` call
- Note that it's the same syntax as for objects, only the Class is held as `process`.

Example:

```
type_transition sshd_t shell_exec_t:process user_t;
```

- ... which means that if a process in the `sshd_t` domain runs an executable of type `shell_exec_t` (a shell, most likely) the process will continue in the `user_t` domain.
- For processes, the `type_transition` statement doesn't include the permission.

Type transitions for processes (cont)

- A lot of permissions need to be explicitly declared: The transition itself, reading and running the executable, and much more
- The `domain_auto_trans` macro includes the type transition statement and a lot of relevant permissions (such as allowing a pipe run between the two relevant domains)
- So instead of the previous example, we may want to go:

```
domain_auto_trans(sshd_t, shell_exec_t, user_t);
```

- In the absence of a matching transition rule, the executable will run without changing the domain. That requires the `execute_no_trans` permission

Type sets and class sets

- A set can be put where a single type or class would normally appear, as long as it makes sense (to whom?)
- Curly brackets '{' and '}' with space-delimited elements mean "for each element"
- The tilde character preceding an expression indicates the complement of the set
- The asterisk * represents all types or classes
- A minus sign preceding an element, within a curly brackets expressions reduces the element from the set
- Examples:

```
allow unconfined_t mytype_t:file { read getattr };
allow unconfined_t mytype_t:file * ;
```

role declaration

```
role ROLE types TYPE;
```

- Meaning “it’s legal for a process context with role ROLE to be in the domain TYPE”
- Sets can be used for the type, but not for the role
- For a list of types currently known by the kernel: `seinfo -r`
- An attempt to enter a domain with an unauthorized role, will cause an “invalid context” error.
- Example:

```
role unconfined_r types mytype_t ;
```

Constraints

- Every permission request must obey all constraints currently active in the kernel
- Shouldn't be necessary in a policy module
- Since it isn't so relevant, we'll just take an example:

```
constrain process transition ( u1 == u2 or t1 == privuser );
constrain process transition ( r1 == r2 or t1 == privrole );
constrain dir_file_class_set { create relabelto relabelfrom }
( u1 == u2 or t1 == privowner );
```

- There's `mlsconstraint` too, which constrains MLS-related permissions (this issue is barely documented)

Declaring types

```
type identifier attributelist ;
```

- This declares the type with the name `identifier`
- The `attributelist` is optional.
- … and the name “attribute” is a misnomer. It’s more like a means for grouping types.
- Examples:

```
type mytype_t;
type crond_t, domain, privuser, privrole, privfd, privowne
```

- Given the type declaration above, if the attribute `privuser` is used where the syntax expected a type, this will include several types, including `crond_t`
- Same goes for `domain`, `privrole`, `privfd` and `privowner`

Declaring attributes and typeattribute

- If you want your own attributes (in a module?) they need to be declared:

```
attribute myattributename;
```

- Also, it's possible to give a type an attribute in a separate statement:

```
typeattribute mytype_t theattribute;
```

Writing an SELinux module

What an SELinux module is

- Just another bunch of declarations and rules injected into the kernel
- Can be unloaded
- Usually covers the security rules for a certain application

A simple module generation strategy

- Define a type for the application's executable
- Define another type, which will be the domain in which the application runs
- The latter type will also be used for files used by the application
- Since the process runs in a domain not defined elsewhere, every possible access to existing objects is denied by default
- Run the application while the system is run in permissive mode. Accesses that would be denied are logged
- Use `audit2allow` to create rules which match the denial log messages
- Tune the rules as necessary

Pros and cons of this method

Pros:

- Easy
- Doesn't require previous awareness of all permissions necessary (and they are oh so many)
- Tight restriction

Cons:

- Covers only what the application did during the test run
- Risk of inserting an unrelated rule by mistake, and opening a security hole

Getting started

- Create a directory to work in
- Make a symbolic link to the development makefile
  `ln -s /usr/share/selinux/devel/Makefile`
- If you don't have that makefile, your development package may be installed elsewhere or not at all.
- Prepare an initial module source file with a `.te` suffix

Getting started (cont.)

- Open a shell window with root privileges, and follow log messages:

```
tail -f /var/log/audit/audit.log | \
     grep -E '^type=(AVC|SELINUX\_ERR)'
```

- `AVC` messages will occur when permissions are denied
- `SELINUX_ERR` messages involve attempts to break role and user restrictions.
- In permissive mode these operation are completed anyhow
- If the audit daemon is off, these messages will go to `/var/log/messages`

Module header

Example:

```
module haifux 1.0.0;

require {
  type unconfined_t;
  class process { transition sigchld };
  class file { read x_file_perms };
}
```

- The first line declares the module's name and version
- The `require` clause indicates which the types and permissions (per class) the module expects to already exist (prior to its loading)

Required vs. defined types

- We have to tell the compiler which types we define, and which already exist.
- If we use a type without defining or requiring it, we get a compilation error like
  `haifux.te":24:ERROR 'unknown type haifux_exec_t' at token ';' on line 1028`
- Or if a class is missing:
  `haifux.te":26:ERROR 'unknown class process' at token ';' on line 1030:`
- Or a permission is missing in the class declarations:
  `haifux.te":45:ERROR 'permission sigchld is not defined for class process' at token ';' on line 1049:`

Required vs. defined types (cont.)

- On the other hand, if we required a class which doesn't exist (possibly because we invented it) the module's load will fail with something like:
  `libsepol.print_missing_requirements: haifux's global requirements were not met: type/attribute haifux_t`
- And if we defined a type which we should have required (it already exists):
  `libsepol.scope_copy_callback: unconfined: Duplicate declaration in module: type/attribute unconfined_t`

A minimal module

Let's start with `haifux.te` as follows:

```
module haifux 1.0.0;

require {
  type unconfined_t;
  class process transition;
}

type haifux_t;
type haifux_exec_t;

role unconfined_r types haifux_t;

type_transition unconfined_t haifux_exec_t : process haifux_t;
```

A minimal module (cont.)

- It defines two new types, `haifux_t` and `haifux_exec_t`.
- It also tells the SELinux core, that if a process in the `unconfined_t` domain runs an executable of whose type is `haifux_exec_t`, the process should continue in the `haifux_t` domain.
- But nothing is allowed for these two types, so they are are both completely useless.

Compiling and loading

- In order to compile the module, run `make` in the working directory
- Some files are generated (well discuss them later)
- The module's binary has a `.pp` suffix
- In order to load the module, run `make load` as root. Be patient – this can take half a minute or so.
- `make clean` does what you'd expect

Our test application

This is `hello.c`, which will compiled into `hello`

```
#include <stdio.h>

int main() {
  printf("Hello, world\n");

  return 0;
}
```

Sort-of explains itself, doesn't it?

Let's try it out

```
[root@rouge]# setenforce 0
[root@rouge]# chcon -t haifux_exec_t hello
[root@rouge]# setenforce 1
[root@rouge]# ./hello
bash: ./hello: Permission denied
[root@rouge]# setenforce 0
[root@rouge]# ./hello
Hello, world
[root@rouge]#
```

Remarks on the session

- `setenforce` switches between permissive mode (value 0) and enforced mode (value 1)
- The type of `hello` was set with `chcon`, which is good enough for trying things out
- The execution of `hello` was denied, since we have no permissions on its type
- To get an idea of how bad things are, go `grep haifux /var/log/audit/audit.log | less`
- Most entries were created during permissive mode. On enforcing mode, things stopped on the first denial.

Setting the permissions straight

- We start over in a new working directory
- Remember to symlink to the makefile
- Let `audit2allow` write the rules for us, based upon the permission denials:

  ```
  grep haifux /var/log/audit/audit.log | \
  audit2allow -m haifux > haifux.te
  ```
- Insert the type declarations from the "minimal module" into the one generated by `audit2allow` and remove their appearance in the `require` clause.
- It's necessary to filter relevant log entries, or the module will open doors to anything attempted on the system
- `grep` is a simple solution in our case

Setting the permissions straight (cont.)

- Review the new rules file *carefully*
- Compile and load like before
- Everything should run well now in enforcement mode
- Now let's do this on Firefox. (Hint: The application will lose it)
- To make it safer, we'll work on a copy of the executable.

Other files in the working directory

- The `make` command created three files
- `haifux.pp` is the module's compiled binary
- `haifux.if` Is generated empty, but could contain code fragment for helping with the `require` clause (does it work?)
- `haifux.fc` contains information about which files must have what context. `make install` will make sure these contexts are permanent (survive relabeling).
- The `.fc` files resemble the format of the `file_context`. A typical line would be:

```
/home/eli/myapp.sh -- gen_context(system_u:object_r:myapp_exec_t,s0)
```

The jail effect

- The process can't escape from its domain, unless explicitly permitted to
- If we start it with a type of our own, such a permission can't exist without our knowledge
- If the process runs another executable, it will run under the same domain (given the permissions, `execute_no_trans` in particular
- Or we can require a transition to another domain we created
- Processes in neither domains can't touch anything unless we explicitly permitted that

Using macros

- The example policies come with a lot of macros, which bundle declarations and rules to form a group that makes sense to humans
- Some of the macros are documented in "Configuring the SELinux policy" by the NSA and elsewhere
- Automatic module generation utilities are most likely to use macros
- They can be found in the policy source files

Macros: The greatest hits

- `domain_auto_trans(sshd_t, shell_exec_t, user_t)` – automatic domain transition with the permissions included
- `file_type_auto_trans(sshd_t, tmp_t, sshd_tmp_t)` – type transition for files, permissions included

Compiling and loading without make

- `m4 mymodule-with-macros.te > mymodule.te` (If there are macros to open)
- `checkmodule -M -m mymodule.te -o mymodule.mod`
- `semodule_package -o mymodule.pp -m mymodule.mod`
- `semodule -i mymodule.pp`
- The `semodule` command loads the module binary, and must be run as root.
- If the module is already loaded, it will be updated.

Compiling and loading without make (cont)

A few remarks:

- The `make` compilation involves stardard macros automagically
- Even worse, the `m4` command above does not know about SELinux-specific macros. They are best copied into the module itself.
- Remember that a macro must be defined before (in the code) it's used.
- Remove the module: `semodule -r mymodule` (as root)

Wrap-up

Random list of command-line utilities

apol, seaudit, sediffx, seaudit-report, sechecker, sediff, seinfo, sesearch, findcon, replcon, indexcon
avcstat, getenforce, getsebool, matchpathcon, selinuxconlist, selinuxdefcon, selinuxenabled, setenforce, togglesebool

Where to look for relevant files

- /selinux
- /usr/share/selinux/devel/
- /etc/selinux
- In the policy source bundle (which may be difficult to find)

References

- Configuring the SELinux Policy (Stephen Smalley, NSA)

  `http://www.nsa.gov/SeLinux/papers/policy2.pdf`
- Security-Enhanced Linux User Guide

  `http://mdious.fedorapeople.org/drafts/html/index.html`
- Red Hat SELinux Guide

  `http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/selinux-guide/`
- Google is your friend

Thank you

Questions?