Security Enhanced Linux (SELinux)

Sunday, October 14, 2018 12:14 PM

Overview

Security Enhanced Linux (SELinux) is a *mandatory access control (MAC)* mechanism for the Linux kernel that is implemented *on top* of the Linux traditional permission-based DAC mechanism (meaning **both the DAC and MAC must approve an action for it to be carried out**.)
SELinux was originally developed by the NSA as a Linux kernel patch, and presented at *the 2.5 Linux Kernel Summit*. This presentation interested Linus, which suggested its generalization to the *Linux Security Modules (LSM)* framework, which was indeed developed by a joint effort of NSA and others, after which both the LSM framework and SELinux (reimplemented as a *LSM module* [3]) was merged into the mainline Linux kernel.

This page, in-most, will not distinct between the Linux and Android implementation of this mechanism, but will refer to both simply as SELinux. The research done on this page is based mostly on Android sources, but the concepts and implementation of both mechanisms are extremely similar, with several changes that will be discussed in the SEAndroid page.

The SELinux Policy

SELinux relies on a *systemwide*, *fine-grained* security policy termed *SELinux policy* that is loaded on system boot from user-space by reading the binary policy file `/sepolicy` to the parsed kernel-space data-structures (*the policy's in-memory representation.)*

The SELinux policy is comprised of three main concepts*: subjects, objects* and *actions. Subjects* are the *active actors* (i.e. processes) that perform *actions* (e.g., *read*, *write, signal* etc.) on *objects* (e.g., *files, sockets, processes, etc.)* The action is carried out in dependence of the security policy and the SELinux mode: *disabled* (no policy loaded and as such all actions are approved,) *permissive* (policy violations are only logged and such all actions are approved,) or *enforcing* (any policy violation disallows the action.)

Both the subjects and the objects are in-memory instances of kernel objects. The SELinux policy cannot specify rules for these runtime instances directly; it must regard to statically defined fields instead. These fields are assigned to the objects and are collectively regarded as the object's *security context.* The *security context* (or *label*) is comprised of four colon-delimited fields: *user identifier, role, type*, and an optional *MLS security range*. The security context of processes can be displayed by specifying the -Z option to the *ps* command; notice the colon-delimited security contexts:

```
# ps -Z
LABEL                          USER     PID   PPID   NAME
u:r:init:s0❶                   root     1     0      /init
u:r:kernel:s0                  root     2     0      kthreadd
u:r:kernel:s0                  root     3     2      ksoftirqd/0
--snip--
u:r:healthd:s0❷                root     175   1      /sbin/healthd
u:r:servicemanager:s0❸         system   176   1      /system/bin/
servicemanager
```

SELinux supports two main MAC types: *type enforcement (TE)* and *multi-level security (MLS.)* In the core of the type enforcement MAC is the *type* field of the security context. A *type* is a unique identifier of an object type that is defined as a unique string in the policy source files. **Android uses a fixed SELinux user identifier, role and security range and uses the *type enforcements (TE) MAC.***

The SELinux policy is comprised of textual source files under the `/system/sepolicy`. These are later compiled onto the *binary policy file `/sepolicy`*.

Type Enforcement Rules

Access Vector Rules

The type enforcement MAC policy is based on *access vector rules*. **SELinux prevents access unless explicitly allowed**, and therefore the policy sources are comprised of `allow` and `neverallow` rules. An *access vector rule* is used to specify the permission set that a certain *subject type* (a specific process) is *allowed* (or *never allowed)* over a *target object type*. These rules can also specify auditing instructions, but this is of less interest to us.

The format of an access vector rule statement is:

```
rule_name source_type target_type:class perm_set;
```

Where rule_name is either `allow`, `dontaudit`, `auditallow`, or `neverallow`; `class` is used to associate the target object type to a class of types in order to deduct its set of defined permissions and `perm_set` is a sub-set of the defined permissions of the target object type's class that this rule applies to.

In order to generalize access vector rules, since there is a lot (~100) object types defined in a policy, two new concepts are introduced: *classes* and *attributes*.

A *class* is used to group *types* with a similar *permission set* (also called *access vectors*). It is declared using the `class` keyword in the *`security_classes`* {11} file. This file contains, for example, the `process`, `socket` and `tcp_socket` classes (snipped):

```
class process

# network-related classes
class socket
class tcp_socket
class udp_socket
class rawip_socket
```

The permission set of each class is defined in the *access_vectors* [{12}] file using the `class` or `common` (to enable permission inheritance) keywords (snipped):

```
common socket
{
# inherited from file
    ioctl
    read
    write
    create
# socket-specific
    bind
    connect
    listen
    accept
}

class tcp_socket
inherits socket
{
    connectto
    newconn
    acceptfrom
    node_bind
    name_connect
}
```

An *attribute* is used to group various types to create generic access vector rules. It can be used in place of the `source_type` or `target_type` units in the access vector rule declaration. Attributes are defined in the *attributes* [{13}] file (snipped):

```
# All types used for devices.
attribute dev_type;

# All types used for processes.
attribute domain;

# All types used for filesystems.
attribute fs_type;
```

An attribute with significance importance is the *domain* attribute. It groups all processes and is used to define generic rules for them. These rules are defined in the *domain.te* [{14}] file (snipped):

```
# Rules for all domains.

# Allow reaping by init.
allow domain init:process sigchld;

# read any sysfs symlinks
allow domain sysfs:lnk_file read;

# Root fs.
allow domain rootfs:dir search;
allow domain rootfs:lnk_file read;
```

Type Transition Rules

A *type transition* rule is used *for specialization* of types. It defines a transition for an object that was created by a specific domain from its original type a sub-type. *Note that there is no concrete inheritance relationships between the type and sub-type, and these terms are only used conceptually to illustrate the use and need for type transitions.*

The format of a type transition rule statement is:

```
type_transition actor source_type:class new_type;
```

It begins with the `type_transition` keyword and is followed by the `actor` that dictates the *domain* that created the object; `source_type` and `class` of the newly created object and the `new_type` of the object after this transition is applied.

As an example, the *wpa* daemon uses this keyword to transition its sockets from the wifi_data_file socket to the specialize wpa_socket socket.

```
# Create a socket for receiving info from wpa
type_transition wpa wifi_data_file:dir wpa_socket "sockets";
```

Domain Transition Rules

Domain transition rules are type transition rules that are used to transition a newly created process object from its original file type (e.g., `mediaserver_exec`) to its process type (defined with a domain attribute.) These are declared using the `init_deamon_domain()` macro. As an example, let's look on the mediaserver process defined in the *mediaserver.te* [{15}] file:

```
# mediaserver - multimedia daemon
type mediaserver, domain, domain_deprecated;
type mediaserver_exec, exec_type, file_type;

typeattribute mediaserver mlstrustedsubject;

init_daemon_domain(mediaserver)
```

The `init_deamon_domain()` macro is defined in the *`te_macros`* {16} file using the `domain_auto_trans()` macro:

```
#####################################
# init_daemon_domain(domain)
# Set up a transition from init to the daemon domain
# upon executing its binary.
define(`init_daemon_domain', `
domain_auto_trans(init, $1_exec, $1)
tmpfs_domain($1)
')
```

The `domain_auto_trans()` macro calls the `domain_trans()` macro to set *allow* access vector rules that are required to allow the transition and uses the `type_transition` keyword under-the-hood:

```
#####################################
# domain_auto_trans(olddomain, type, newdomain)
# Automatically transition from olddomain to newdomain
# upon executing a file labeled with type.
#
define(`domain_auto_trans', `
# Allow the necessary permissions.
domain_trans($1,$2,$3)
# Make the transition occur by default.
type_transition $1 $2:process $3;
')
```

For clarity and brevity purposes, I manually expended the relevant parts of these macros. The final result is something of the like:

```
# Old domain may exec the file and transition to the new domain.
allow init mediaserver_exec:file { getattr open read execute };
allow init mediaserver:process transition;
# New domain is entered by executing the file.
allow mediaserver mediaserver_exec:file { entrypoint open read execute getattr };

type_transition init mediaserver_exec:process mediaserver;
```

Where mediaserver_exec is the type of the mediaserver executable file on the filesystem. This can be seen in the *`file_contexts`* {17} file:

```
/init                   u:object_r:init_exec:s0
/system/bin/mediaserver u:object_r:mediaserver_exec:s0
```

In conclusion, since the *init* process is the domain that creates all processes in Android, the domain transition macros allow it the required permissions on the target executable (typed `xxx_exec`, e.g., `mediaserver_exec`,) allow *init* the transition permission from the process class on the new domain, allow the new domain the required permission on its own executable file, and automatically enable the transition using the `type_transition` keyword where the actor that created the object is *init* and the transition is from the executable type to the new domain.

The Binary Policy File `/sepolicy`

The binary policy file is little-endian. This a hexdump of the beginning of this file (taken from a Galaxy S8 Nougat build):

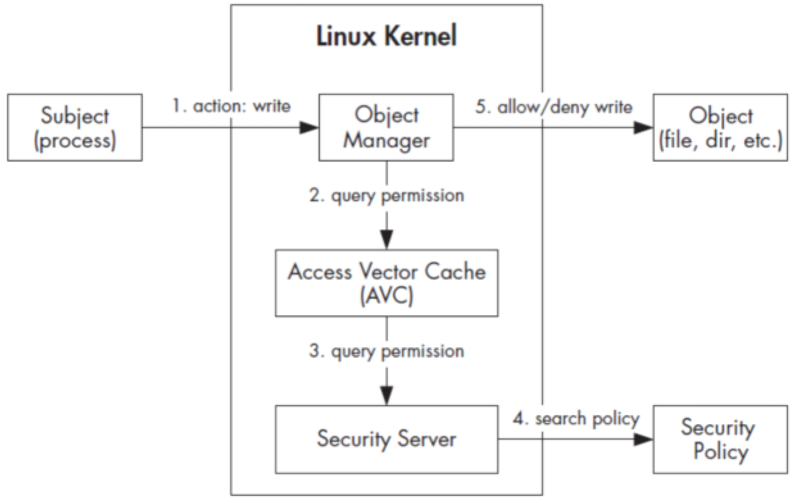
```
8C FF 7C F9 08 00 00 00 53 45 20 4C 69 6E 75 78   Œÿ|ù....SE Linux
1E 00 00 00 01 00 00 00 08 00 00 00 07 00 00 00   ................
40 00 00 00 40 00 00 00 01 00 00 00 00 00 00 00   @...@...........
03 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
00 00 00 00 03 00 00 00 03 00 00 00 06 00 00 00   ................
02 00 00 00 16 00 00 00 16 00 00 00 73 6F 63 6B   ............sock
65 74 06 00 00 00 0A 00 00 00 61 70 70 65 6E 64   et........append
```

The first two values are two sanity values. The first four bytes of the policy binary file is the magic value 0xf97cff8c. The second sanity value the SELinux magic string "SE Linux". The second four bytes correspond to the string length which is followed by the string itself. We can see it clearly in the right side of the hexdump.

The next four bytes that represent the database version. We can see that the version in our binary file is 30 (0x1E).

SELinux Architecture

The SELinux architecture consists of four main components: *object managers (OM),* an *access vector cache (AVC)*, a *security server*, and a *security policy*. When a subject asks to perform an action on an *SELinux object* (for example, when a process tries to read a file), the associated object manager queries the AVC to see if the attempted action is allowed. If the AVC contains a cached security decision for the request, the AVC returns it to the OM, which enforces the decision by allowing or denying the action (steps 1, 2, and 5). If the cache does not contain a matching security decision, the AVC contacts the security server, which makes a security decision based on the currently loaded policy and returns it to the AVC, which caches it. The AVC in turn returns it to the OM, which ultimately enforces the decision (steps 1, 2, 3, 4, and 5).



Filesystem Security Labeling

We can view the security contexts of files using the `ls -Z` command (snipped):

```
dreamlte:/ $ ls / -Z
u:object_r:cache_file:s0        cache
u:object_r:configfs:s0          config
u:object_r:system_data_file:s0  data
u:object_r:device:s0            dev
u:object_r:rootfs:s0            etc
u:object_r:rootfs:s0            file_contexts.bin
u:object_r:init_exec:s0         init
u:object_r:rootfs:s0            init.rc
u:object_r:rootfs:s0            lib
u:object_r:tmpfs:s0             mnt
u:object_r:proc:s0              proc
u:object_r:rootfs:s0            root
u:object_r:rootfs:s0            sbin
u:object_r:rootfs:s0            sdcard
u:object_r:rootfs:s0            sepolicy
u:object_r:storage_file:s0      storage
u:object_r:sysfs:s0             sys
u:object_r:system_file:s0       system
```

The security labeling of files is mediated using the *super block* structure of each filesystem. The `super_block` is a per-filesystem structure that holds various characteristics of the filesystem and is shared for all child files (`inodes`) of that mount point. A copy of the super block for non-volatile filesystems is saved on disk.

The `super_block` structure holds a SELinux-specific security field of the `superblock_security_struct` structure:

```
struct superblock_security_struct {
    struct super_block *sb;     /* back pointer to sb object */
    u32 sid;            /* SID of file system superblock */
    u32 def_sid;            /* default SID for labeling */
    u32 mntpoint_sid;       /* SECURITY_FS_USE_MNTPOINT context for files */
    unsigned short behavior;    /* labeling behavior */
    unsigned short flags;       /* which mount options were specified */
    struct mutex lock;
    struct list_head isec_head;
    spinlock_t isec_lock;
};
```

The most important field for our purpose is the `behavior` field which determines the *labeling behavior* of files under this filesystem. There are six different labeling behaviors:

```
#define SECURITY_FS_USE_XATTR       1 /* use xattr */
#define SECURITY_FS_USE_TRANS       2 /* use transition SIDs, e.g. devpts/tmpfs */
#define SECURITY_FS_USE_TASK        3 /* use task SIDs, e.g. pipefs/sockfs */
#define SECURITY_FS_USE_GENFS       4 /* use the genfs support */
#define SECURITY_FS_USE_NONE        5 /* no labeling support */
#define SECURITY_FS_USE_MNTPOINT    6 /* use mountpoint labeling */
#define SECURITY_FS_USE_NATIVE      7 /* use native label support */
```

The labeling behavior is determined by the `security_fs_use()` function which consults the policy database's `fs_use` object contexts(`policydb.ocontexts[OCON_FSUSE]`.) This is defined in the *fs_use* policy source file:

```
# Label inodes via getxattr.
fs_use_xattr yaffs2 u:object_r:labeledfs:s0;
fs_use_xattr jffs2 u:object_r:labeledfs:s0;
fs_use_xattr ext2 u:object_r:labeledfs:s0;
fs_use_xattr ext3 u:object_r:labeledfs:s0;
fs_use_xattr ext4 u:object_r:labeledfs:s0;
fs_use_xattr xfs u:object_r:labeledfs:s0;
fs_use_xattr btrfs u:object_r:labeledfs:s0;
fs_use_xattr f2fs u:object_r:labeledfs:s0;
fs_use_xattr squashfs u:object_r:labeledfs:s0;

# Label inodes from task label.
fs_use_task pipefs u:object_r:pipefs:s0;
fs_use_task sockfs u:object_r:sockfs:s0;

# Label inodes from combination of task label and fs label.
# Define type_transition rules if you want per-domain types.
fs_use_trans devpts u:object_r:devpts:s0;
fs_use_trans tmpfs u:object_r:tmpfs:s0;
fs_use_trans devtmpfs u:object_r:device:s0;
fs_use_trans shm u:object_r:shm:s0;
fs_use_trans mqueue u:object_r:mqueue:s0;
```

As we can see, the default non-volatile file system ext4 uses extended attributes where pipes and sockets are labeled after the task that created them.

The *genfs* labeling method is used in filesystems that cannot support a persistent label mapping or use another fixed labeling behavior like transition SIDs or task SIDs. It is defined in the *genfs_contexts* policy source file (snipped):

```
# Label inodes with the fs label.
genfscon rootfs / u:object_r:rootfs:s0
# proc labeling can be further refined (longest matching prefix).
genfscon proc / u:object_r:proc:s0
genfscon proc /iomem u:object_r:proc_iomem:s0
genfscon proc /meminfo u:object_r:proc_meminfo:s0
genfscon proc /net u:object_r:proc_net:s0
genfscon proc /cpuinfo u:object_r:proc_cpuinfo:s0
genfscon proc /sys/kernel/modprobe u:object_r:usermodehelper:s0
genfscon proc /sys/kernel/randomize_va_space u:object_r:proc_security:s0
genfscon proc /sys/net u:object_r:proc_net:s0
genfscon proc /sys/vm/mmap_min_addr u:object_r:proc_security:s0

# selinuxfs booleans can be individually labeled.
genfscon selinuxfs / u:object_r:selinuxfs:s0
genfscon cgroup / u:object_r:cgroup:s0
# sysfs labels can be set by userspace.
genfscon sysfs / u:object_r:sysfs:s0
genfscon debugfs / u:object_r:debugfs:s0
genfscon tracefs / u:object_r:debugfs_tracing:s0
genfscon fuse / u:object_r:fuse:s0
genfscon configfs / u:object_r:configfs:s0
genfscon sdcardfs / u:object_r:sdcardfs:s0
genfscon usbfs / u:object_r:usbfs:s0
```

As is evident from the source file, most of the known in-memory virtual filesystems, like *sysfs, proc, selinuxfs* and so on, use *genfs* support. That is in contradiction to the *socket* and *pipe* virtual filesystems that use task SID labeling (as seen in `fs_use`.)

When labeling newly-created files, the security is filled in before first use. This is done by an LSM hook on `d_instantiate()`. The callback, `selinux_d_instantiate()` calls `inode_doinit_with_dentry()` to initialize the security attributes of the `inode`. This function consults the super block of that `inode`, and acts in accordance to its labeling behavior.

SELinux Initialization

Overview

"Nothing [in Biology] Makes Sense Except in the Light of Evolution" ~Christian Theodosius Dobzhansky

I believe it is of outmost importance to deeply understand the initialization flow in order to understand anything better. It is from the very start from which we begin our journey. It would make everything clear, and is the best starting point for almost any research or documentation.

I have chosen to inspect the initialization sequence of SEAndroid. The following code path is taken from AOSP (for the sources to *init* and *libselinux*) and Galaxy S8 Nougat (for kernel sources.) This should suffice to get the point across for SELinux initialization as well.

SELinux initialization splits into two distinct flows: the *early initialization flow*, that occurs during the loading of the kernel as a *initcall*, and *security server initialization and policy loading flow*, which is initiated by *init* and initializes most of the kernel-side implementation of SELinux, including filling up its various data structures and after which SELinux is ready to be used in its full power.

The early initialization flow is responsible for setting up basic support to label processes and objects before the policy is loaded and the security server is fully initialized. It begins by allocating the `security` structure for the init task, setting its SID to *kernel.* It follows by initializing the AVC and registering SELinux's LSM callbacks.

The security server and policy loading initialization sequence starts with the `main()` function under `/system/core/init/init.cpp` {1}. `main()` is the entry point of the *init* process - the first process initialized in the system. SELinux initialization begins there; this shows how early in the OS boot sequence it occurs. *init* maps the binary policy file `/sepolicy` and uses *selinuxfs* to access the kernel-side SELinux code to perform the kernel-side initialization and policy loading procedure by writing the memory-mapped binary policy data to `/sys/fs/selinux/load`.

The main part of SELinux initialization is parsing the binary policy data into the `policydb` structure.

When the kernel SELinux code finished loading the policy, *init* performs a `restorecon()` call and re-executes in order to continue executing under the *init_exec* domain.

init is the only process that can load policies. This is assured by two facts:

1. *init* is the first user space code that is executed in the system and it initializes SELinux and loads the SELinux policy upon execution.
2. The (now loaded) SELinux policy contains a rule that allows *init* and *init* only to load SELinux policies. This could be seen in the policy sources under `system/sepolicy/domain.te`:

```
# Only init should be able to load SELinux policies.
# The first load technically occurs while still in the kernel domain,
# but this does not trigger a denial since there is no policy yet.
# Policy reload requires allowing this to the init domain.
neverallow { domain -init } kernel:security load_policy;
```

In some versions of Android, even *init* cannot load policies after the initial policy has been loaded. This can be seen in this commit:

```
- # Only init should be able to load SELinux policies.
- # The first load technically occurs while still in the kernel domain,
- # but this does not trigger a denial since there is no policy yet.
- # Policy reload requires allowing this to the init domain.
- neverallow { domain -init } kernel:security load_policy;
-
- # Only init and the system_server can set selinux.reload_policy 1
- # to trigger a policy reload.
- neverallow { domain -init -system_server } security_prop:property_service set;
+ # Once the policy has been loaded there shall be none to modify the policy.
+ # It is sealed.
+ neverallow * kernel:security load_policy;
```

SELinux Initialization Sequence

Early Initialization

```
/* SELinux requires early initialization in order to label
   all processes and objects when they are created. */
security_initcall(selinux_init);
```

`selinux_init()` {8} begins by calling `security_module_enable("selinux")`.

```
static __init int selinux_init(void)
{
    if (!security_module_enable("selinux")) {
```

This method simply compares the string passed to it and the `chosen_lsm` variable.

```c
/**
 * security_module_enable - Load given security module on boot ?
 * @module: the name of the module
 *
 * Each LSM must pass this method before registering its own operations
 * to avoid security registration races. This method may also be used
 * to check if your LSM is currently loaded during kernel initialization.
 *
 * Return true if:
 *  -The passed LSM is the one chosen by user at boot time,
 *  -or the passed LSM is configured as the default and the user did not
 *   choose an alternate LSM at boot time.
 * Otherwise, return false.
 */
int __init security_module_enable(const char *module)
{
    return !strcmp(module, chosen_lsm);
}
```

The `chosen_lsm` variable comes from `CONFIG_DEFAULT_SECURITY`.

```c
/* Boot-time LSM user choice */
static __initdata char chosen_lsm[SECURITY_NAME_MAX + 1] =
    CONFIG_DEFAULT_SECURITY;
```

`selinux_init()` continues by calling `cred_init_security()` to initialize the security for the init task, setting its initial SID to `SECINITSID_KERNEL`.

```c
/*
 * initialise the security for the init task
 */
static void cred_init_security(void)
{
    struct cred *cred = (struct cred *) current->real_cred;
    struct task_security_struct *tsec;
#ifdef CONFIG_RKP_KDP
    tsec = &init_sec;
    tsec->bp_cred = cred;
#else
    tsec = kzalloc(sizeof(struct task_security_struct), GFP_KERNEL);
    if (!tsec)
        panic("SELinux:  Failed to initialize initial task.\n");
#endif
    tsec->osid = tsec->sid = SECINITSID_KERNEL;
    cred->security = tsec;
}
```

`selinux_init()` continues by creating the `kmem_cache` structures that will later be used to allocate the specified security structures for the `inode` and `file` structures.

```c
sel_inode_cache = kmem_cache_create("selinux_inode_security",
                    sizeof(struct inode_security_struct),
                    0, SLAB_PANIC, NULL);
file_security_cache = kmem_cache_create("selinux_file_security",
                    sizeof(struct file_security_struct),
                    0, SLAB_PANIC, NULL);
```

`selinux_init()` continues by calling `avc_init()` {9} to initialize the *access vector cache (AVC)*. It then finishes by calling `security_add_hooks()` to register `selinux_hooks` to the LSM's `security_hook_heads[]` {7} array.

```c
avc_init();

security_add_hooks(selinux_hooks, ARRAY_SIZE(selinux_hooks));
```

Security Server Initialization and Policy Loading

The code path for SELinux initialization follows.

`main()` has much code with little relevance to our purposes. I will only note that prior to initializing SELinux, `main()` initializes mounts *sysfs* at /sys. This is important since *selinuxfs*, the interface through which the user-space *libselinux* library communicate with the kernel-side SELinux implementation for initialization and policy loading, is mounted as a sub-directory of *sysfs* at */sys/fs/selinux*.

1. Skipping to the relevant part of `main()`, it calls `selinux_initialize()` to set up SELinux:

```c
// Set up SELinux, including loading the SELinux policy if we're in the kernel domain.
selinux_initialize(is_first_stage);
```

2. `selinux_initalize()` is executed within the security context of the `kernel` domain in the first policy load only. It uses this fact in order to distinguish the first initialization. In the case of the first initialization, which is the only case I will cover, it begins by loading the SELinux policy through `selinux_android_load_policy()` which is exposed by *libselinux* {2}:

```
if (in_kernel_domain) {
    INFO("Loading SELinux policy...\n");
    if (selinux_android_load_policy() < 0) {
        ERROR("failed to load policy: %s\n", strerror(errno));
        security_failure();
    }
```

3. `selinux_android_load_policy()` begins by mounting *selinuxfs*:

```
int selinux_android_load_policy(void)
{
    const char *mnt = SELINUXMNT;
    int rc;
    rc = mount(SELINUXFS, mnt, SELINUXFS, 0, NULL);
```

4. It then issues a call to an internal function, `selinux_android_load_policy_helper()`, in which the rest of the initialization loading is implemented. This distinction is important to increase code re-use, as this internal function implements the policy loading logic that is used both in policy loading and in policy reloading.

```
return selinux_android_load_policy_helper(false);
```

5. `selinux_android_load_policy_helper()` begins by opening and `mmap()`ing the policy file at */sepolicy*. It continues by calling `security_load_policy()` on this map.
6. security_load_policy() opens the virtual file */sys/fs/selinux/load* and writes the policy binary data onto it.
7. Transitioning into the kernel-side implementation of *selinuxfs* {3} we inspect the filesystem initialization code.

```
static struct tree_descr selinux_files[] = {
    [SEL_LOAD] = {"load", &sel_load_ops, S_IRUSR|S_IWUSR},
    [SEL_ENFORCE] = {"enforce", &sel_enforce_ops, S_IRUGO|S_IWUSR},
    [SEL_CONTEXT] = {"context", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_ACCESS] = {"access", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_CREATE] = {"create", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_RELABEL] = {"relabel", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_USER] = {"user", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_POLICYVERS] = {"policyvers", &sel_policyvers_ops, S_IRUGO},
    [SEL_COMMIT_BOOLS] = {"commit_pending_bools", &sel_commit_bools_ops, S_IWUSR},
    [SEL_MLS] = {"mls", &sel_mls_ops, S_IRUGO},
    [SEL_DISABLE] = {"disable", &sel_disable_ops, S_IWUSR},
    [SEL_MEMBER] = {"member", &transaction_ops, S_IRUGO|S_IWUGO},
    [SEL_CHECKREQPROT] = {"checkreqprot", &sel_checkreqprot_ops, S_IRUGO|S_IWUSR},
    [SEL_REJECT_UNKNOWN] = {"reject_unknown", &sel_handle_unknown_ops, S_IRUGO},
    [SEL_DENY_UNKNOWN] = {"deny_unknown", &sel_handle_unknown_ops, S_IRUGO},
    [SEL_STATUS] = {"status", &sel_handle_status_ops, S_IRUGO},
    [SEL_POLICY] = {"policy", &sel_policy_ops, S_IRUGO},
    /* last one */ {""}
};
ret = simple_fill_super(sb, SELINUX_MAGIC, selinux_files);
if (ret)
    goto err;
```

8. To understand what code will execute upon writing to the */load* file, we inspect `sel_load_ops`.

```
static const struct file_operations sel_load_ops = {
    .write      = sel_write_load,
    .llseek     = generic_file_llseek,
};
```

9. So when `security_load_policy()` issued the `write()` syscall, control flow has been directed onto `sel_write_load()`. This function begins by assuring the task has the `load_policy` permission:

```
length = task_has_security(current, SECURITY__LOAD_POLICY);
if (length)
    goto out;
```

On the first load, there is no policy loaded and as such this check will succeed. At future policy reloads, this check will succeed in the case that the current process is under the *init* domain only, as we specified before.

10. It continues by copying the policy data that has been sent from usermode and calling `security_load_policy()` {4}.

```
if (copy_from_user(data, buf, count) != 0)
    goto out;

length = security_load_policy(data, count);
```

11. `security_load_policy()` is the service that is exposed by the *Security Server* for policy loading.

```
/**
 * security_load_policy - Load a security policy configuration.
 * @data: binary policy data
 * @len: length of data in bytes
 *
 * Load a new set of security policy configuration data,
 * validate it and convert the SID table as necessary.
 * This function will flush the access vector cache after
 * loading the new policy.
 */
int security_load_policy(void *data, size_t len)
```

It begins by checking the global variable `ss_initialized` which indicates whether the Security Server (SS) has already been initialized. The case where it isn't initialized is the only one we are concerned about and it begins by initializing the *access vector table cache* by calling `avtab_cache_init()`, after which it can read the policy DB by calling `policydb_read()` {5}:

```
if (!ss_initialized) {
    avtab_cache_init();
    rc = policydb_read(&policydb, fp);
```

12. `policydb_read()` is responsible to read and parse the binary policy data into the policy database structure.

```
/*
 * Read the configuration data from a policy database binary
 * representation file into a policy database structure.
 */
int policydb_read(struct policydb *p, void *fp)
```

It begins by initializing the policy database by calling `policydb_init()` to initialize the `policydb` structure. This in turn initializes the *access vector table,* the *roles*, the *conditional policy database*, and *various* `ebitmaps`.

13. The first fields of the binary policy file are a magic value, the SELinux magic string, and the policy version. The magic value and string are compared as a sanity check. The version is used to decide upon version-dependent features.
14. The next four bytes that are read represent some bitmap. These decides whether MLS is to be configured, and whether to allow or reject unknown.
15. `policydb_read()` continues by initializing *the symbol table*.
16. It then reads the *access vector table* by calling avtab_read().
17. It continues to read the *security contexts* to the `ocontexts` member of the `policydb` structure by calling `ocontext_read()`.
20. It continues by filling up the *genfs* data by calling `genfs_read()`.
21. `policydb_read()` finishes by filling up the type -> attribute reverse mapping `type_attr_map_array`. This mapping maps the source type to its type attributes map. This type attribute map is an `ebitmap` where each bit represents the target type, and it is set if there exists a *type enforcement rule* for this (source type, target type) pair and unset otherwise. If the version supports it, it reads the attributes `ebitmap` from the binary database. Otherwise, type enforcement rules are unsupported and only the degenerated case (where the type is able to transition to itself) is added to the attributes (`e` is the type attributes `ebitmap` for type `i`):

```
ebitmap_init(e);
if (p->policyvers >= POLICYDB_VERSION_AVTAB) {
    rc = ebitmap_read(e, fp);
    if (rc)
        goto bad;
}
/* add the type itself as the degenerate case */
rc = ebitmap_set_bit(e, i, 1);
```

22. `security_load_policy()` continues by setting the current mapping by calling `selinux_set_mapping()`. `selinux_set_mapping()` is responsible for allocating and filling up the current SELinux mapping. `security_load_policy()` supplies it with the already loaded policy database, a pointer for the `current_mapping[]` and `current_mapping_size` global variables as output parameters, and the global `security class mapping secclass map[]`:

```
rc = selinux_set_mapping(&policydb, secclass_map,
                 &current_mapping,
                 &current_mapping_size);
```

It begins by scanning `secclass_map[]` (which it refers to as the *input mapping*) and allocating an identically sized *output mapping* which will be assigned to the `current_mapping[]` global variable later.
Then it starts looping the input mapping to store raw class and permission values in the output mapping:

```
/* Store the raw class and permission values */
j = 0;
while (map[j].name) {
    struct security_class_mapping *p_in = map + (j++);
    struct selinux_mapping *p_out = out_map + j;
```

It begins by assigning the policy value for the current security class. This is performed by a call to `string_to_security_class()` which simply searches it in the classes symbol table of the loaded policy database:

```
u16 string_to_security_class(struct policydb *p, const char *name)
{
    struct class_datum *cladatum;

    cladatum = hashtab_search(p->p_classes.table, name);
    if (!cladatum)
        return 0;

    return cladatum->value;
}
```

It then continues by assigning permissions:

```
 k = 0;
 while (p_in->perms && p_in->perms[k]) {
     /* An empty permission string skips ahead */
     if (!*p_in->perms[k]) {
         k++;
         continue;
     }
     p_out->perms[k] = string_to_av_perm(pol, p_out->value,
                         p_in->perms[k]);
     if (!p_out->perms[k]) {
         printk(KERN_INFO
                "SELinux:  Permission %s in class %s not defined in policy.\n",
                p_in->perms[k], p_in->name);
         if (pol->reject_unknown)
             goto err;
         print_unknown_handle = true;
     }

     k++;
 }
 p_out->num_perms = k;
```

23. In conclusion, `selinux_set_mapping()` translates the string-based `secclass_map[]` of all the supported classes and permissions to the value-based `current_mapping[]` by the values defined in the currently loaded policy database.
24. `security_load_policy()` continues by loading the `sidtab` via a `policydb_load_isids()` call. This function walks the initial SIDs linked-list in the `policydb` structure (`p->ocontexts[OCON_ISID]`) and calls `sidtab_insert()` to insert it into the *SID table.*
25. `security_load_policy()` sets the `ss_initialized` global variable to 1, indicating that the *Security Server* is initialized!
    Getting back to `sel_write_load()`, it continues by updating the SELinux virtual filesystem by filling the following directories: *class*, *booleans* and *policy_capabilities*.
26. Getting back to `selinux_inialize()`, after policy loading has completed, it finished initialization by setting the *enforcing* mode to the one given in the command line (returned by `selinux_is_enforcing()`) if this is allowed, and if it differs from the current *enforcing* mode (returned by `security_getenforce()`):

```
bool kernel_enforcing = (security_getenforce() == 1);
bool is_enforcing = selinux_is_enforcing();
if (kernel_enforcing != is_enforcing) {
    if (security_setenforce(is_enforcing)) {
        ERROR("security_setenforce(%s) failed: %s\n",
              is_enforcing ? "true" : "false", strerror(errno));
        security_failure();
    }
}
```

27. `selinux_intialize()` has now complete, and we go back to `main()`. The *init* process has to transition from its previous `kernel` domain into the `init` domain. It does so by using the `restorecon()` which wraps the `selinux_android_restorecon()` function

that is exposed by *libselinux*:

```
// If we're in the kernel domain, re-exec init to transition to the init domain now
// that the SELinux policy has been loaded.
if (is_first_stage) {
    if (restorecon("/init") == -1) {
        ERROR("restorecon failed: %s\n", strerror(errno));
        security_failure();
    }
    char* path = argv[0];
    char* args[] = { path, const_cast<char*>("--second-stage"), nullptr };
    if (execv(path, args) == -1) {
        ERROR("execv(\"%s\") failed: %s\n", path, strerror(errno));
        security_failure();
    }
}
```

The `restorecon()` function restores the security context of files to the ones specified in the *file_contexts.bin* file (compiled from the *file_contexts* source file). In issuing it, *init* has changed the security context of its executable. That being said, it is not sufficient to change the domain of the current process. To do so, it must re-execute the /init executable. The new process will inherit the domain from the security context that has just been set to it, which now is the `init_exec` domain, as is evident from the source *file_contexts*:

```
/init                    u:object_r:init_exec:s0
```

This is what assures that the *init* process can issue additional (re)load policy requests successfully.

28. `selinux_android_restorecon()` wraps the `selinux_android_restorecon_common()` function is also responsible to load the security context of files on it first call:

```
__selinux_once(fc_once, file_context_init);
```

Which calls `selinux_android_file_context_handle()` to read the */file_contexts.bin* file and store these labels in the global variable `fc_sehandle`. This will be later used by the internal function `restorecon_sb()` to restore the security context of files by modifying the *security.selinux extended attribute* using the `lsetxattr` system call:

```
lsetxattr(path, XATTR_NAME_SELINUX, context, strlen(context) + 1,
  0);
```

29. The `lsetxattr` system call is defined in the xattr.c file:

```
SYSCALL_DEFINE5(lsetxattr, const char __user *, pathname,
        const char __user *, name, const void __user *, value,
        size_t, size, int, flags)
{
    return path_setxattr(pathname, name, value, size, flags, 0);
}
```

30. The code path leads to both LSM hooks and the *Extended Verification Module (EVM)* to perform further validation before setting the extended attribute.
31. Assuming all security checks has passed, the attribute is set using the __vfs_setxattr_noperm() function. This internally calls the `setaxttr()` function pointer that is exposed by the underlaying implementation of the virtual file system that holds the file.

```
inode->i_op->setxattr(dentry, name, value, size, flags)
```

For non-volatile file systems that support extended attributes, such as Android's default file system, ext4, this call commits the new extended attributes to disk.

32. Finally, `main()` issues the `restorecon()` function on directories that were created before setting up SELinux and the *init* domain correctly, in order to fix their security contexts:

```
// These directories were necessarily created before initial policy load
// and therefore need their security context restored to the proper value.
// This must happen before /dev is populated by ueventd.
NOTICE("Running restorecon...\n");
restorecon("/dev");
restorecon("/dev/socket");
restorecon("/dev/__properties__");
restorecon("/property_contexts");
restorecon_recursive("/sys");
```

The Access Check Flow

We will give as an example the complete access check flow for file renaming under SELinux, starting with `vfs_rename()` in *namei.c* {6}. Other checks have very similar flow, and this example should be sufficient for an overall understanding of the underlaying code path, but a read of the underlaying *Linux Security Modules (LSM)* mechanism is recommended.

1. `vfs_rename()` is called with the following parameters:

```
 * vfs_rename - rename a filesystem object
 * @old_dir:    parent of source
 * @old_dentry: source
 * @new_dir:    parent of destination
 * @new_dentry: destination
 * @delegated_inode: returns an inode needing a delegation break
 * @flags:  rename flags
```

The only relevant code in this function for our purposes is the call to the LSM function `security_inode_rename()` {7}:

```
error = security_inode_rename(old_dir, old_dentry, new_dir, new_dentry,
                  flags);
if (error)
    return error;
```

If `security_inode_rename()` will return a non-zero error code `vfs_rename()` will terminate and the rename operation will abort.

2. `security_inode_rename()` wraps around `call_int_hook()`:

```
return call_int_hook(inode_rename, 0, old_dir, old_dentry,
                  new_dir, new_dentry);
```

3. `call_int_hook()` is a macro that traversals the list of callbacks to this particular hook (`security_hook_heads[].inode_rename`) and calls each one in order:

```
#define call_int_hook(FUNC, IRC, ...) ({            \
    int RC = IRC;                       \
    do {                        \
        struct security_hook_list *P;           \
                                \
        list_for_each_entry(P, &security_hook_heads.FUNC, list) { \
            RC = P->hook.FUNC(__VA_ARGS__);      \
            if (RC != 0)                \
                break;              \
        }                           \
    } while (0);                        \
    RC;                         \
})
```

It returns upon the first callback that returns a non-zero error code (thus disallows the operation) and thus allows the operation (by returning 0) only if all callbacks for this hook returns 0.

In SELinux the hook for `inode_rename()` is defined as `selinux_inode_rename()` in *security/selinux_n/hooks.c* {8}:

```
RKP_RO_AREA static struct security_hook_list selinux_hooks[] = {
    ...
    LSM_HOOK_INIT(inode_rename, selinux_inode_rename),
    ...
}
```

4. `selinux_inode_rename()` wraps around `may_rename()`:

```
static int selinux_inode_rename(struct inode *old_inode, struct dentry *old_dentry,
                struct inode *new_inode, struct dentry *new_dentry)
{
#ifdef CONFIG_RKP_KDP ···
#endif  /* CONFIG_RKP_KDP */
    return may_rename(old_inode, old_dentry, new_inode, new_dentry);
}
```

5. `may_rename()` implements the real logic of SELinux's renaming access check.
    a. It begins by fetching the security context of the current task by calling `current_sid()`:

```c
/*
 * get the subjective security ID of the current task
 */
static inline u32 current_sid(void)
{
    const struct task_security_struct *tsec = current_security();

    return tsec->sid;
}
```

This internally calls `current_security()` which just returns the `security` field of the `cred` member of the current task.

b. It then issues a series of policy enforcement checks via the *access vector cache (AVC)* by calling upon `avc_has_perm()` {9}.
Renaming a file or directory is a complex action and is thus composed of multiple sub-actions, which translates to calls to the AVC, each with different target objects and requested permissions, but there are some commonalities to them all:
   - The source SID in all of these calls is the security context of the current task (computed at a.)
   a. The target SID changes, and is fetched from the `inode` structure of the target object.
   This information is stored and fetched from the extended attributes of the filesystem.
   The security class is derived from the target object, which is most likely either a `SECCLASS_DIR` for directories (e.g. in policy enforcement checks done on the parent directories, or when the target `inode` is a directory) or `SECCLASS_FILE` (when the source and target `inode` structures represents simple files.)
   The requested permissions vary; these are the possible permissions that could be requested: `DIR__REMOVE_NAME`, `DIR__SEARCH`, `FILE__RENAME`, `DIR__REPARENT`, `DIR__ADD_NAME`, `DIR__RMDIR`, and `FILE__UNLINK`.

The complete algorithm of `may_rename()` can therefore by summarized as such:

   i. Assure the current task can search (i.e. list files) the parent directory of the file that is pending renaming (i.e. the *old file*) and can remove names from it (i.e. deleting the old file node from it.)

```c
rc = avc_has_perm(sid, old_dsec->sid, SECCLASS_DIR,
          DIR__REMOVE_NAME | DIR__SEARCH, &ad);
if (rc)
    return rc;
```

   ii. Assure the current task has rename permission on the old file

```c
rc = avc_has_perm(sid, old_isec->sid,
          old_isec->sclass, FILE__RENAME, &ad);
if (rc)
    return rc;
```

   iii. If the user is trying to move a directory to a different path, assure he is able to do so (*reparent* that directory)

```c
if (old_is_dir && new_dir != old_dir) {
    rc = avc_has_perm(sid, old_isec->sid,
              old_isec->sclass, DIR__REPARENT, &ad);
    if (rc)
        return rc;
}
```

   iv. Assure that the current task can add new names, and search from the new directory. If the new file already exists, assure that it can also delete files from that directory.

```c
ad.u.dentry = new_dentry;
av = DIR__ADD_NAME | DIR__SEARCH;
if (d_is_positive(new_dentry))
    av |= DIR__REMOVE_NAME;
rc = avc_has_perm(sid, new_dsec->sid, SECCLASS_DIR, av, &ad);
if (rc)
    return rc;
```

   v. If the new file already exists, assure that the current task can delete that file.

```c
if (d_is_positive(new_dentry)) {
    new_isec = d_backing_inode(new_dentry)->i_security;
    new_is_dir = d_is_dir(new_dentry);
    rc = avc_has_perm(sid, new_isec->sid,
              new_isec->sclass,
              (new_is_dir ? DIR__RMDIR : FILE__UNLINK), &ad);
    if (rc)
        return rc;
}
```

6. `avc_has_perm()` is the main interface exposed by the AVC to perform policy enforcement checks. It is defined as such:

```c
/**
 * avc_has_perm - Check permissions and perform any appropriate auditing.
 * @ssid: source security identifier
 * @tsid: target security identifier
 * @tclass: target security class
 * @requested: requested permissions, interpreted based on @tclass
 * @auditdata: auxiliary audit data
 *
 * Check the AVC to determine whether the @requested permissions are granted
 * for the SID pair (@ssid, @tsid), interpreting the permissions
 * based on @tclass, and call the security server on a cache miss to obtain
 * a new decision and add it to the cache.  Audit the granting or denial of
 * permissions in accordance with the policy.  Return %0 if all @requested
 * permissions are granted, -%EACCES if any permissions are denied, or
 * another -errno upon other errors.
 */
int avc_has_perm(u32 ssid, u32 tsid, u16 tclass,
         u32 requested, struct common_audit_data *auditdata)
```

It performs auditing if needed, and then calls upon `avc_has_perm_noaudit()` to perform the internal logic of policy enforcement checks.

```c
int avc_has_perm(u32 ssid, u32 tsid, u16 tclass,
         u32 requested, struct common_audit_data *auditdata)
{
    struct av_decision avd;
    int rc, rc2;

    rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, 0, &avd);

    rc2 = avc_audit(ssid, tsid, tclass, requested, &avd, rc, auditdata, 0);
    if (rc2)
        return rc2;
    return rc;
}
```

7. `avc_has_perm_noaudit()` first tries to lookup the access vector in the cache by calling `avc_lookup()`:
   ```c
   node = avc_lookup(ssid, tsid, tclass);
   ```
   If it is unsuccessful, it means that the access vector for this operation isn't cached. Thus, it calls upon `avc_compute_av()` to consult the *security server* with the policy decision-making procedure for this operation by calling `security_compute_av()` and cache the result it in the AVC for future lookups by calling `avc_insert()`.
   ```c
   if (unlikely(!node))
       node = avc_compute_av(ssid, tsid, tclass, avd, &xp_node);
   ```
8. `security_compute_av()` {10} is a service that the security server exposes to the AVC and is the actual policy decision-making code.
   ```c
   /**
    * security_compute_av - Compute access vector decisions.
    * @ssid: source security identifier
    * @tsid: target security identifier
    * @tclass: target security class
    * @avd: access vector decisions
    * @xperms: extended permissions
    *
    * Compute a set of access vector decisions based on the
    * SID pair (@ssid, @tsid) for the permissions in @tclass.
    */
   void security_compute_av(u32 ssid,
                u32 tsid,
                u16 orig_tclass,
                struct av_decision *avd,
                struct extended_perms *xperms)
   ```
   It begins by searching the security context for the source SID and target SID:
   ```c
   scontext = sidtab_search(&sidtab, ssid);
   ```
   This is done by calling `sidtab_search()` with the global `sidtab` hashmap. This function wraps around `sidtab_search_core()`.
9. `sidtab_search_core()` begins by performing a search in the `sidtab` hashmap:

```
hvalue = SIDTAB_HASH(sid);
cur = s->htable[hvalue];
while (cur && sid > cur->sid)
    cur = cur->next;
```

If it doesn't find a suitable entry it remaps the SID to the *unlabeled SID.*

```
if (cur == NULL || sid != cur->sid || cur->context.len) {
    /* Remap invalid SIDs to the unlabeled SID. */
    sid = SECINITSID_UNLABELED;
    hvalue = SIDTAB_HASH(sid);
    cur = s->htable[hvalue];
    while (cur && sid > cur->sid)
        cur = cur->next;
    if (!cur || sid != cur->sid)
        return NULL;
}
```

10. Now, after `security_compute_av()` has retrieved the security context for both the source and target objects, it continues by retrieving the real, policy values for the target class by calling `unmap_class()` which retrieves it from the mapped values (i.e. `current_mapping[tclass].value`)

```
tclass = unmap_class(orig_tclass);
if (unlikely(orig_tclass && !tclass)) {
    if (policydb.allow_unknown)
        goto allow;
    goto out;
}
```

11. It then finally has all the resources to compute the access vectors and extended permissions by calling `context_struct_compute_av()`:

```
context_struct_compute_av(scontext, tcontext, tclass, avd, xperms);
```

Which is defined as such:

```
/*
 * Compute access vectors and extended permissions based on a context
 * structure pair for the permissions in a particular class.
 */
static void context_struct_compute_av(struct context *scontext,
                    struct context *tcontext,
                    u16 tclass,
                    struct av_decision *avd,
                    struct extended_perms *xperms)
```

It first retrieves the `class` struct by accessing the policy's call value to struct mapping. it is an array that is indexed by `class value - 1`:

```
tclass_datum = policydb.class_val_to_struct[tclass - 1];
```

It then computes the final access vector by combining all *the access vector rules* defined in the policy that match the source and target types and the combination of all their *attributes*. The attributes that are defined for each type is retrieved from the [*extensible bitmaps* (`ebitmap`)] `policydb.type_attr_map_arrray`:

```
/*
 * If a specific type enforcement rule was defined for
 * this permission check, then use it.
 */
avkey.target_class = tclass;
avkey.specified = AVTAB_AV | AVTAB_XPERMS;
sattr = flex_array_get(policydb.type_attr_map_array, scontext->type - 1);
BUG_ON(!sattr);
tattr = flex_array_get(policydb.type_attr_map_array, tcontext->type - 1);
BUG_ON(!tattr);
ebitmap_for_each_positive_bit(sattr, snode, i) {
    ebitmap_for_each_positive_bit(tattr, tnode, j) {
```

For each *(source type, target type)* pair, where *source/target type* is either the original type or one of its attributes, the function searches for all the access vector rules defined in the policy using `avtab_search_node()` and `avtab_search_node_next()`. This search yields `avtab_node` structures, each representing a single access vector rule: either *an allow, auditallow, dontaudit*, or rules that defines extended permissions, respectively:

```
avkey.source_type = i + 1;
avkey.target_type = j + 1;
for (node = avtab_search_node(&policydb.te_avtab, &avkey);
      node;
      node = avtab_search_node_next(node, avkey.specified)) {
    if (node->key.specified == AVTAB_ALLOWED)
        avd->allowed |= node->datum.u.data;
    else if (node->key.specified == AVTAB_AUDITALLOW)
        avd->auditallow |= node->datum.u.data;
    else if (node->key.specified == AVTAB_AUDITDENY)
        avd->auditdeny &= node->datum.u.data;
    else if (xperms && (node->key.specified & AVTAB_XPERMS))
        services_compute_xperms_drivers(xperms, node);
}
```

12. `context_struct_compute_av()` finishes by checking the conditional av table for additional permissions, removing any permissions prohibited by a constraint (which also includes the MLS policy), and other end-cases I will not cover here.
13. After fetching the access vector (either because it existed in the cache, or after the security server performed its policy decision-making procedure) `avc_has_perm_noaudit()` performs a simple comparison on the access vector and decides whether the action is permitted.

```
denied = requested & ~(avd->allowed);
if (unlikely(denied))
    rc = avc_denied(ssid, tsid, tclass, requested, 0, 0, flags, avd);
```

14. `avc_denied()` implements the final relevant logic. It mainly assures that SELinux is enforcing (if AVC_STRICT is defined as 1 it just disallow the operation; otherwise, it checks the selinux_enforcing variable and the AVD flags to determine if it is permissive) and therefore decides whether to disallow this operation (enforcing) or allow it and update the AVC node to grant that operation for later access (permissive.)

The following implementation of `avc_denied()` is taken from Linux 4.4:

```
static noinline int avc_denied(u32 ssid, u32 tsid,
                               u16 tclass, u32 requested,
                               u8 driver, u8 xperm, unsigned flags,
                               struct av_decision *avd)
{
        if (flags & AVC_STRICT)
                return -EACCES;

        if (selinux_enforcing && !(avd->flags & AVD_FLAGS_PERMISSIVE))
                return -EACCES;

        avc_update_node(AVC_CALLBACK_GRANT, requested, driver, xperm, ssid,
                        tsid, tclass, avd->seqno, NULL, flags);
        return 0;
}
```

Samsung has disabled this function entirely by simply returning an error, thus there is no meaning for permissiveness whatsoever and SELinux enforces every action. This was documented in [5]:

```
avc_denied.isra.0
MOV                 W0, #0xFFFFFFF3
RET
; End of function avc_denied.isra.0
```

15. This result is propagated back to `avc_has_perm() -> may_rename() -> selinux_inode_rename() -> security_inode_rename()` and finally to `vfs_rename()`, with which our journey completes.

Data Structures

The Policy Database `policydb`

policydb is the in-memory representation of the parsed binary SELinux policy. It is of the following structure:

```c
struct policydb {
    int mls_enabled;
    /* symbol tables */
    struct symtab symtab[SYM_NUM];
    ...
    /* symbol names indexed by (value - 1) */
    struct flex_array *sym_val_to_name[SYM_NUM];
    /* class, role, and user attributes indexed by (value - 1) */
    struct class_datum **class_val_to_struct;
    struct role_datum **role_val_to_struct;
    struct user_datum **user_val_to_struct;
    struct flex_array *type_val_to_struct_array;
    /* type enforcement access vectors and transitions */
    struct avtab te_avtab;
    /* role transitions */
    struct role_trans *role_tr;
    /* file transitions with the last path component */
    /* quickly exclude lookups when parent ttype has no rules */
    struct ebitmap filename_trans_ttypes;
    /* actual set of filename_trans rules */
    struct hashtab *filename_trans;
    /* bools indexed by (value - 1) */
    struct cond_bool_datum **bool_val_to_struct;
    /* type enforcement conditional access vectors and transitions */
    struct avtab te_cond_avtab;
    /* linked list indexing te_cond_avtab by conditional */
    struct cond_node *cond_list;
    /* role allows */
    struct role_allow *role_allow;
    /* security contexts of initial SIDs, unlabeled file systems,
       TCP or UDP port numbers, network interfaces and nodes */
    struct ocontext *ocontexts[OCON_NUM];
    /* security contexts for files in filesystems that cannot support
       a persistent label mapping or use another
       fixed labeling behavior. */
    struct genfs *genfs;
    /* range transitions table (range_trans_key -> mls_range) */
    struct hashtab *range_tr;
    /* type -> attribute reverse mapping */
    struct flex_array *type_attr_map_array;
    struct ebitmap policycaps;
    struct ebitmap permissive_map;
    /* length of this policy when it was loaded */
    size_t len;
    unsigned int policyvers;
    unsigned int reject_unknown : 1;
    unsigned int allow_unknown : 1;
    u16 process_class;
    u32 process_trans_perms;
};
```

The Extendible Bit Map `ebitmap`

An extensible bitmap is a bitmap that supports an arbitrary number of bits. Extensible bitmaps are used to represent sets of values, such as types, roles, categories, and classes.

Each extensible bitmap is implemented as a linked list of bitmap nodes, where each bitmap node has an explicitly specified starting bit position within the total bitmap.

```c
struct ebitmap_node {
    struct ebitmap_node *next;
    unsigned long maps[EBITMAP_UNIT_NUMS];
    u32 startbit;
};

struct ebitmap {
    struct ebitmap_node *node;  /* first node in the bitmap */
    u32 highbit;    /* highest position in the total bitmap */
};
```

The Symbol Table `symtab`

The symbol table, `symtab`, is actually an array of symbol tables of SYM_NUM size. The `policydb` structure has macros that define shorted access to each symbol table within that array. These are the symbol table within `symtab`:

```
#define p_commons symtab[SYM_COMMONS]
#define p_classes symtab[SYM_CLASSES]
#define p_roles symtab[SYM_ROLES]
#define p_types symtab[SYM_TYPES]
#define p_users symtab[SYM_USERS]
#define p_bools symtab[SYM_BOOLS]
#define p_levels symtab[SYM_LEVELS]
#define p_cats symtab[SYM_CATS]
```

Internally, each of these is a simple hash map from a string to a value. This value is a pointer to a struct that represent the specific type of that table. For example, the value for `p_classes` is of type `class_datum*`.

```
struct symtab {
    struct hashtab *table;  /* hash table (keyed on a string) */
    u32 nprim;       /* number of primary names in table */
};
```

The Access Vector Table

The access vector table is a table of *access vector keys* of structure `avtab_key`.

```
struct avtab_key {
    u16 source_type;    /* source type */
    u16 target_type;    /* target type */
    u16 target_class;   /* target object class */
#define AVTAB_ALLOWED        0x0001
#define AVTAB_AUDITALLOW     0x0002
#define AVTAB_AUDITDENY      0x0004
#define AVTAB_AV         (AVTAB_ALLOWED | AVTAB_AUDITALLOW | AVTAB_AUDITDENY)
#define AVTAB_TRANSITION     0x0010
#define AVTAB_MEMBER         0x0020
#define AVTAB_CHANGE         0x0040
#define AVTAB_TYPE       (AVTAB_TRANSITION | AVTAB_MEMBER | AVTAB_CHANGE)
/* extended permissions */
#define AVTAB_XPERMS_ALLOWED     0x0100
#define AVTAB_XPERMS_AUDITALLOW 0x0200
#define AVTAB_XPERMS_DONTAUDIT  0x0400
#define AVTAB_XPERMS         (AVTAB_XPERMS_ALLOWED | \
                AVTAB_XPERMS_AUDITALLOW | \
                AVTAB_XPERMS_DONTAUDIT)
#define AVTAB_ENABLED_OLD    0x80000000 /* reserved for used in cond_avtab */
#define AVTAB_ENABLED        0x8000 /* reserved for used in cond_avtab */
    u16 specified;  /* what field is specified */
};
```

As we can see, this structure defines some rule between a *source type, target type and target class.* The rule type is specified by the *specified* member. These could either be *access vector rules* (e.g. *allow* rules) *type rules* (e.g. *type transition* rules) or *extended permissions.*

The Object Contexts Array `ocontext`

The `ocontexs` field of the policy database structure is an array of `ocontext` linked lists.

```
/* object context array indices */
#define OCON_ISID  0    /* initial SIDs */
#define OCON_FS    1    /* unlabeled file systems */
#define OCON_PORT  2    /* TCP and UDP port numbers */
#define OCON_NETIF 3    /* network interfaces */
#define OCON_NODE  4    /* nodes */
#define OCON_FSUSE 5    /* fs_use */
#define OCON_NODE6 6    /* IPv6 nodes */
#define OCON_NUM   7

/* security contexts of initial SIDs, unlabeled file systems,
   TCP or UDP port numbers, network interfaces and nodes */
struct ocontext *ocontexts[OCON_NUM];
```

Each `ocontext` linked-list is of the following structure:

```c
/*
 * The configuration data includes security contexts for
 * initial SIDs, unlabeled file systems, TCP and UDP port numbers,
 * network interfaces, and nodes.  This structure stores the
 * relevant data for one such entry.  Entries of the same kind
 * (e.g. all initial SIDs) are linked together into a list.
 */
struct ocontext {
    union {
        char *name; /* name of initial SID, fs, netif, fstype, path */
        struct {
            u8 protocol;
            u16 low_port;
            u16 high_port;
        } port;      /* TCP or UDP port information */
        struct {
            u32 addr;
            u32 mask;
        } node;      /* node information */
        struct {
            u32 addr[4];
            u32 mask[4];
        } node6;        /* IPv6 node information */
    } u;
    union {
        u32 sclass;  /* security class for genfs */
        u32 behavior;  /* labeling behavior for fs_use */
    } v;
    struct context context[2];  /* security context(s) */
    u32 sid[2]; /* SID(s) */
    struct ocontext *next;
};
```

SELinux Mapping

`current_mapping` is defined as a global array of `selinux_mapping` structures:

```c
struct selinux_mapping {
    u16 value; /* policy value */
    unsigned num_perms;
    u32 perms[sizeof(u32) * 8];
};

static struct selinux_mapping *current_mapping;
static u16 current_mapping_size;
```

This array is indexed by the class value.

The Security Class Mapping `security_class_mapping`

The security class mapping `secclass_map[]` is defined as a global array of `security_class_mapping` structures:

```c
/* Class/perm mapping support */
struct security_class_mapping {
    const char *name;
    const char *perms[sizeof(u32) * 8 + 1];
};

extern struct security_class_mapping secclass_map[];
```

It is also instantiated statically at *classmap.h* (attaching a snip of the definition, see file for full definition):

```
struct security_class_mapping secclass_map[] = {
    { "security",
      { "compute_av", "compute_create", "compute_member",
        "check_context", "load_policy", "compute_relabel",
        "compute_user", "setenforce", "setbool", "setsecparam",
        "setcheckreqprot", "read_policy", NULL } },
    { "process",
      { "fork", "transition", "sigchld", "sigkill",
        "sigstop", "signull", "signal", "ptrace", "getsched", "setsched",
        "getsession", "getpgid", "setpgid", "getcap", "setcap", "share",
        "getattr", "setexec", "setfscreate", "noatsecure", "siginh",
        "setrlimit", "rlimitinh", "dyntransition", "setcurrent",
        "execmem", "execstack", "execheap", "setkeycreate",
        "setsockcreate", NULL } },
    { "system",
      { "ipc_info", "syslog_read", "syslog_mod",
        "syslog_console", "module_request", "module_load", NULL } },
    { "capability",
      { "chown", "dac_override", "dac_read_search",
        "fowner", "fsetid", "kill", "setgid", "setuid", "setpcap",
        "linux_immutable", "net_bind_service", "net_broadcast",
        "net_admin", "net_raw", "ipc_lock", "ipc_owner", "sys_module",
        "sys_rawio", "sys_chroot", "sys_ptrace", "sys_pacct", "sys_admin"
        "sys_boot", "sys_nice", "sys_resource", "sys_time",
        "sys_tty_config", "mknod", "lease", "audit_write",
        "audit_control", "setfcap", NULL } },
    { "filesystem",
      { "mount", "remount", "unmount", "getattr",
        "relabelfrom", "relabelto", "associate", "quotamod",
        "quotaget", NULL } },
    { "file",
      { COMMON_FILE_PERMS,
        "execute_no_trans", "entrypoint", NULL } },
    { "dir",
      { COMMON_FILE_PERMS, "add_name", "remove_name",
        "reparent", "search", "rmdir", NULL } },
    { "binder", { "impersonate", "call", "set_context_mgr", "transfer",
              NULL } },
    { NULL }
  };
```

This means the kernel holds a static map of all supported classes and permissions. The strings here are identical to the ones in the source (textual) policy definition files, and remain plain-text strings in the binary policy file, as could be seen in the hexdump taken from */sepolicy*.

Source Code References

All code snippets here was done on a Galaxy S8 Nougat.

1. *system/core/init/init.cpp*
   - main()
2. *external/libselinux/src/android.c*
   - selinux_android_load_policy()
3. *security/selinux_n/selinuxfs.c*
   - selinux_files
   - sel_load_ops
   - sel_write_load()
4. *security/selinux_n/ss/services.c*
   - security_load_policy()
   - security_compute_av()
5. *security/selinux_n/ss/policydb.c*
   - policydb_read()
6. *fs/namei.c*
   - vfs_rename()
7. *security/security.c*
   - security_hook_heads[]
   - security_inode_rename()
   - call_int_hook()

8. *security/selinux_n/hooks.c*
    - `selinux_init()`
    - `selinux_inode_rename()`
    - `may_rename()`
    - `current_sid()`
9. *security/selinux_n/avc.c*
    - `avc_init()`
    - `avc_has_perm()`
    - `security_compute_av()`
    - `avc_denied()`
10. *security/selinux_n/services.c*
    - `security_compute_av()`
11. *system/sepolicy/security_classes*
    - `process`
    - `socket`
    - `tcp_socket`
12. *system/sepolicy/access_vectors*
    - `socket`
    - `tcp_socket`
13. *system/sepolicy/attributes*
14. *system/sepolicy/domain.te*
15. *system/sepolicy/mediaserver.te*
16. *system/sepolicy/te_macros*
    - `init_deamon_domain()`
    - `domain_auto_trans()`
    - `domain_trans()`
17. *system/sepolicy/file_contexts*

External References

1. Android Security Internals
    a. Chapter 1: Android's Security Model
    b. Chapter 12: SELinux
2. Android Internals
3. Implementing SELinux as a Linux Security Module, NSA.
   https://www.nsa.gov/resources/everyone/digital-media-center/publications/research-papers/assets/files/implementing-selinux-as-linux-security-module-report.pdf
4. Security Enhanced (SE) Android - Bringing Flexible MAC to Android
   http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/02_4.pdf
5. Defeating Samsung KNOX with Zero Privilege wp, Blackhat US 2017
   https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf