SELinux Support for Userspace Object Managers

Eamon Walsh
NSA

ewalsh@epoch.ncsc.mil

Initial: May 2004, Last revised: May 2004

Table of Contents

1. Introduction **1**
2. Architecture **1**
3. Tips **2**
4. Example Usage **3**

1. Introduction

The Security-Enhanced Linux project has long focused on implementing mandatory access control using the Flask architecture in the Linux kernel. The sample security server in the kernel is aware of a wide variety of object classes, including processes, files, and sockets. Policy can be written governing access to these objects.

Modern Linux systems, however, have a number of security-relevant userspace daemons and utilities which manage objects and provide services independently of the kernel. Examples include the X Window System server, which manages the display, and the D-BUS daemon, a message-passing utility. Recent research has focused on making these programs "SELinux aware" by having them label internal objects with security contexts and enforce policy over them, querying the kernel to obtain policy decisions.

Starting with version 1.9, libselinux includes a userspace AVC which provides supporting functionality for userspace object managers. This paper gives an overview of its architecture and includes sample code showing proper usage of the library.

The full API for the userspace AVC is documented in the header file `selinux/avc.h`. Additionally, man pages are included with libselinux starting with version 1.13. `avc_init`(3) is a good starting point when using the man pages.

2. Architecture

The SELinux pseudo-filesystem includes a file `access` which is used to obtain policy decisions from the kernel. The libselinux routine `security_compute_av` encapsulates this functionality. However, this routine has the overhead of a kernel trap on each call.

The userspace AVC is essentially a cache built on top of `security_compute_av`. It provides a cleaner interface to the caller, including:

- Optional user-provided callbacks for auditing, memory allocation, and threading.
- A mapping of security contexts to opaque, reference-counted "security ID's" (SIDs).
- Automatic monitoring of permissive vs. enforcing mode.
- Automatic cache flushing on policy changes.
- Tracking of cache statistics.
- Convenience functions for converting between string representations of security classes and access permissions and their actual numeric policy values.

After initialization, a userspace program passes security contexts to `avc_context_to_sid` to obtain SIDs. SIDs are reference-counted; in addition to `avc_context_to_sid`, which increments the reference count, the functions `sidget` and `sidput` increment and decrement the count, respectively. The function `avc_sid_to_context` returns a copy of the context corresponding to a given SID.

Policy decisions are determined using `avc_has_perm`, which takes subject and object SIDs, the object class, and the requested access permissions. The return value of `avc_has_perm` is zero on grant, nonzero (with errno set) otherwise. The function also takes a cache entry reference that speeds cache lookups on repeated queries, and a pointer to supplementary audit data associated with the security class. `avc_has_perm` makes a call to `avc_audit`; use `avc_has_perm_noaudit` if you wish to separate these two actions.

`avc_audit` prints the familiar avc "denied" messages on a policy denial. The default is to print them on standard error. However, userspace programs can provide a printf-style callback to handle the messages themselves — via syslog, for example. Userspace programs can also provide a callback to interpret the extra auditing data passed to `avc_has_perm` and `avc_audit`. This can make the audit messages easier to track and interpret.

Starting with version 2.6.4, the Linux kernel supports netlink notification of policy and enforcing-mode changes. The userspace AVC listens for these notifications and takes the appropriate action (e.g. cache flush) automatically. In the default, single-threaded mode, the userspace AVC must check the netlink socket during each call to `avc_has_perm*`. Performance-critical programs can provide threading and locking callbacks to the userspace AVC which will be used to start a dedicated thread to wait on the socket. The example code below shows how to set up the threading callbacks using the pthread library.

The userspace AVC provides three functions for obtaining statistics. The first two, `avc_av_stats` and `avc_sid_stats`, produce audit messages that indicate the status of the hash tables storing access vectors and SID's, respectively. The messages contain the number of entries, number of hash buckets used, and longest chain of entries in a single bucket. The third statistics function, `avc_cache_stats`, populates an avc_cache_stats structure whose fields describe access vector cache activity (number of lookups performed, hit rate, etc.)

3. Tips

- The context returned by `avc_sid_to_context` must be freed by the caller using `freecon`.

- Remember that `avc_context_to_sid` increments SID reference counts. If you pass the same context three times to this function, the SID for that context will have a count of 3. This behavior supports obtaining SIDs to assign to newly created objects.
- libselinux has facilities for converting from security classes and access vectors into strings and vice versa.
- `avc_has_perm_noaudit` can be used to perform a permission check without auditing. The decision returned by this function can be passed to `avc_audit` to produce the message. See the implementation of `avc_has_perm`.
- When experimenting with new policy, note that `avc_has_perm*` will return -1 with errno set to EINVAL on an invalid security context or security class (this is what `security_compute_av` returns). No audit message is logged in this case.
- If a netlink socket error occurs in single-threaded mode, `avc_has_perm*` will log a message and return with errno set to whatever the socket routine returned. Note that this value might be EACCES (the "normal" errno value for a policy denial).
- If a netlink socket error occurs in threaded mode, the netlink thread will log a message and then terminate. At this point, `avc_has_perm*` will return EINVAL until the userspace AVC is destroyed and reinitialized.
- The userspace AVC produces a log message whenever a netlink notification is processed. Note that in non-threaded mode netlink messages are not processed until the next call to `avc_has_perm*`.
- In addition to SIDs, consider storing an avc_entry_ref structure in each managed object. These structures are passed to `avc_has_perm` and can increase performance on repeated permission checks. Remember to initialize the structures with the `avc_entry_ref_init` macro.
- `avc_cleanup` can be called periodically to free up memory in the userspace AVC.
- `avc_reset` will flush all cached access decisions and reset the userspace AVC's internal statistics. The SID table, however, is not affected. A call to this function is made internally when a netlink policy change notification arrives.

4. Example Usage

The following example application illustrates the use of the userspace AVC. The program reads file pathnames on standard input and checks for read, write, and delete access. The pthread library is used for threading and locking. All output, including audit messages, is printed on standard output.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <pthread.h>
#include <selinux/flask.h>
#include <selinux/selinux.h>
#include <selinux/avc.h>
#include <selinux/av_permissions.h>

/* ---------- auditing callbacks ---------- */
void audit_print(const char *fmt, ...)
```

```
{
    /* we use stdout instead of the default stderr */
    va_list ap;
    va_start(ap, fmt);
    vprintf(fmt, ap);
    va_end(ap);
}

void audit_interp(void *data, security_class_t class,
                  char *buf, size_t buflen)
{
    /* data is a filename */
    snprintf(buf, buflen, (char*)data);
}

/* ---------- threading callbacks ---------- */
void* create_thread_helper(void *arg)
{
    /* arg is the function we need to run */
    void (*run)(void) = (void (*)(void))arg;

    /* set ourself to immediate cancel mode */
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    /* go do our work */
    run();

    /* should never get here */
    return NULL;
}

void* create_thread(void (*run)(void))
{
    int rc;
    pthread_t *t = (pthread_t*)malloc(sizeof(pthread_t));
    if (!t) {
        puts("create_thread: out of memory");
        exit(99);
    }
    /* have the new thread run the helper function above */
    rc = pthread_create(t, NULL, create_thread_helper, (void*)run);
    if (rc) {
        puts("create_thread failed");
        exit(2);
    }
    return t;
}

void stop_thread(void *thread)
{
    int rc = pthread_cancel(*((pthread_t*)thread));
    if (rc) {
        puts("trouble stopping thread");
```

```
        exit(2);
    }
    free(thread);
}

/* ---------- locking callbacks ---------- */
void* alloc_lock(void)
{
    int rc;
    pthread_mutexattr_t pma;
    pthread_mutex_t *m = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
    if (!m) {
        puts("alloc_lock: out of memory");
        exit(99);
    }
    /* set the lock to error checking mode for debugging purposes */
    rc = pthread_mutexattr_init(&pma);
    rc |= pthread_mutexattr_settype(&pma, PTHREAD_MUTEX_ERRORCHECK_NP);
    rc |= pthread_mutex_init(m, &pma);
    rc |= pthread_mutexattr_destroy(&pma);
    if (rc) {
        puts("trouble initializing lock");
        exit(3);
    }
    return m;
}

void get_lock(void *lock)
{
    int rc = pthread_mutex_lock((pthread_mutex_t*)lock);
    if (rc) {
        puts("trouble obtaining lock");
        exit(3);
    }
}

void release_lock(void *lock)
{
    int rc = pthread_mutex_unlock((pthread_mutex_t*)lock);
    if (rc) {
        puts("trouble releasing lock");
        exit(3);
    }
}

void free_lock(void *lock)
{
    int rc = pthread_mutex_destroy((pthread_mutex_t*)lock);
    if (rc) {
        puts("trouble destroying lock");
        exit(3);
    }
    free(lock);
```

```
}

/* ---------- main routine ---------- */
int main (int argc, char **argv) {
    security_context_t scon, fcon;
    security_id_t ssid, fsid;
    char buf[1024];
    struct avc_entry_ref aeref;
    struct avc_cache_stats acs;
    int rc, short_of_memory = 0;

    /* logging callbacks */
    struct avc_log_callback alc = {
        audit_print,
        audit_interp
    };

    /* thread callbacks */
    struct avc_thread_callback atc = {
        create_thread,
        stop_thread
    };

    /* locking callbacks */
    struct avc_lock_callback akc = {
        alloc_lock,
        get_lock,
        release_lock,
        free_lock
    };

    avc_entry_ref_init(&aeref);

    /* use standard malloc/free for the memory callbacks */
    if (avc_init("myprog", NULL, &alc, &atc, &akc) < 0) {
        puts("could not initialize avc");
        exit(1);
    }

    /* get our process security context and a SID for it */
    if (getcon(&scon) < 0) {
        puts("could not get self context");
        exit(5);
    }
    if (avc_context_to_sid(scon, &ssid) < 0) {
        puts("could not get self sid");
        exit(5);
    }

    /* read filenames from stdin */
    while (scanf("%s", buf) != EOF)
    {
        /* force unused cache entries to be freed if necessary */
```

```
        if (short_of_memory)
            avc_cleanup();

        /* get security context and SID for file */
        if (getfilecon(buf, &fcon) < 0) {
            printf("couldn't get file context for '%s'\n", buf);
            continue;
        }
        if (avc_context_to_sid(fcon, &fsid) < 0) {
            printf("could not get file sid for '%s'\n", buf);
            exit(5);
        }

        /* see if we can do some things to file */
        errno = 0;
        rc = avc_has_perm(ssid, fsid, SECCLASS_FILE,
                          FILE__READ | FILE__WRITE | FILE__UNLINK,
                          &aeref, buf);
        if (rc == 0)
            printf("%s: granted\n", buf);
        else if (errno == EACCES)
            printf("%s: denied\n", buf);
        else
            printf("%s: unexpected error: %s\n", buf, strerror(errno));
    }

    /* print out statistics */
    avc_av_stats();
    avc_sid_stats();
    avc_cache_stats(&acs);
    printf("entry_lookups:\t%d\n", acs.entry_lookups);
    printf("entry_hits:\t%d\n", acs.entry_hits);
    printf("entry_misses:\t%d\n", acs.entry_misses);
    printf("entry_discards:\t%d\n", acs.entry_discards);
    printf("cav_lookups:\t%d\n", acs.cav_lookups);
    printf("cav_hits:\t%d\n", acs.cav_hits);
    printf("cav_probes:\t%d\n", acs.cav_probes);
    printf("cav_misses:\t%d\n", acs.cav_misses);

    /* free all AVC resources */
    avc_destroy();

    return 0;
}
```