The need for setuid style functionality in SELinux environments

Fernando Vázquez
University of Vigo
Department of Electronic Technology
email: flvazquez@uvigo.es

Takashi Horie, Toshiharu Harada
NTT DATA CORPORATION
Research and Development Headquarters
email: {horietk, haradats}@nttdata.co.jp

Abstract

Security Enhanced Linux (SELinux) is a software infrastructure that provides the Linux systems with Mandatory Access Control (MAC) capabilities, but a reasonable strict access control policy may render many of the existing application unusable. The latter is not due to SELinux's misbehavior, but to the lack of least privilege considerations in the design of most of the existing applications. For this reason, some of them need to be granted extremely powerful permissions, which could not be safe, or, alternatively, the system administrator can decide to undertake the necessary changes in the code of all the applications, which is likely to be unfeasible. In this paper, the authors propose a solution in which the SELinux kernel is provided with stripped-down `setuid`-like functionalities, that greatly ease the integration of existing applications in a SELinux environment. The access control implications of this approach, as well as the resulting security-usability trade-offs, are also covered in this article.

1 Introduction

As mentioned in the abstract, SELinux is a software infrastructure that provides Linux systems with MAC capabilities. This functionality was implemented into the kernel as a Linux Security Module (LSM), whose security policy is configured from user space using a policy compiler and a loader (to learn the SELinux basics, the documents [1] and [2] are a good start).

One of the basic design decisions of SELinux was the way in which processes can change their security context, operation that is only allowed upon `execve` execution. This restriction was adopted by the SELinux authors because they consider `setuid`-like functions harmful, though, doing this, a somewhat unbalanced security-usability trade-off was implicitly assumed. Thus, applications that were not designed to support least privilege and isolation (or that do not need it) can hardly benefit from SELinux. Further discussion on this important topics is carried out in section 2.

After concluding that features analogical to those offered by the standard `set*uid` functions is necessary, the authors of this paper decided to implement them and test their benefits and shortcomings in a widely used application, such as *Samba*. The modifications made to the Linux kernel (section 4), the SELinux user-space components, such as the policy (section 5) and libraries (section 6), and Samba (section 7) are covered in detail.

Since an in-depth discussion of SELinux, LSM and Samba is beyond the scope of this paper, a list of relevant bibliography on this topics has been included in the *Bibliography* section.

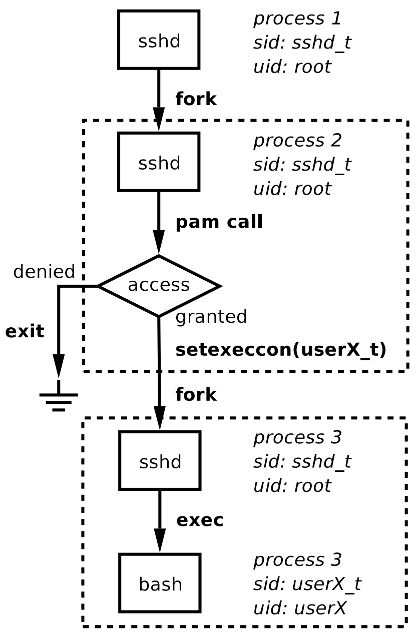
2 Domain transition on exec. Advantages and limitations

2.1 User transitions on *execve*

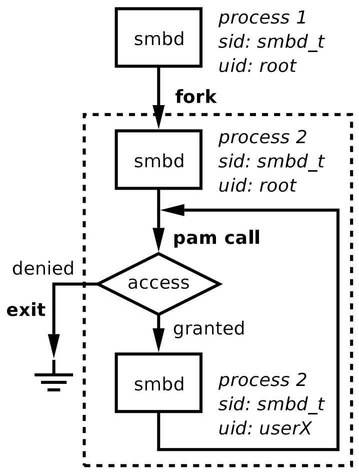
SELinux only supports process Security Identifier (SID) transitions upon program execution so that inheritance of state and the initialization of the process in the new SID can be controlled [3]. This was a design decision of the creators of SELinux, but it imposes very severe limitations, that keep many applications from being able to take advantage of the kernel's new access control capabilities. Figure 1 shows the transition possibilities of *sshd* and *smbd*[1] with the existing SELinux implementation.

As an example, let's consider a scenario in which Samba needs to take advantage of SELinux, so that, in the event of a user session, a dedicated child process should be created in a user-specific domain.

[1]*smbd* is the Samba daemon that implements the session services. The other Samba daemon is *nmbd*, and deals mostly with the Windows network browsing tasks.



(a)



(b)

Figure 1: *(left)* The session shell can transit to a new domain *(right)* Transitions are only allowed on *execve* execution

What should be done?

Before answering this question, a brief explanation of the way in which user sessions are established is Samba is necessary (to better follow the explanation consult figure 1(b)). When *smbd* (*process 1*) receives a connection request, it spawns a child *process 2* to handle the session and, after performing some brief internal accounting, it resumes its listening duties. This way, potential requests are not left unattended during the authentication process. Both the authentication and the eventual session are handled by the child process (*process 2*). Upon successful authentication the child changes its Linux User Identifier (UID) to that of the connecting user, making use of the `seteuid` function. However, it regains root identity, and thus privileges, for several operations, including the authentication of new users. The latter is possible due to the extension of the SMB protocol, that now supports multiple users per session.

Since *smbd* makes no use of `execve`, no domain transitions are possible with the current SELinux implementation. As a consequence, sessions run in the same security context as the parent (i.e. *smbd_t*) and, therefore, upon an eventual failure in the session management code, a malicious user could access any type the main *smbd* process has access to, which, of course, includes other users' data. Being restrained to one single domain context, current SELinux mechanisms cannot be used to further isolate and control user sessions, and this is why this task relies completely on the Samba application itself. But the latter is not a good approach, since user space cannot guarantee security without being backed by adequate Operating System (OS) mechanisms.

If, despite of what has been discussed so far, a system administrator wanted to run every user session on its own security context, the *smbd* code would have to be extensively modified so that it can re-exec itself every time someone authenticates (see figure 2(a)). Another approach would be to split the session management code from *smbd*, creating a separate program that would be executed when someone successfully authenticates. None of these two solutions is an easy task. Furthermore, adding support for SELinux to applications such as *Apache*, or, in general, any application that provides user sessions by itself (without executing an external application for that purpose), would require the implementation of similar changes, which is very likely not to be feasible. Further to this, modifying applications on such a fashion is not good from a software design point of view, since the under-

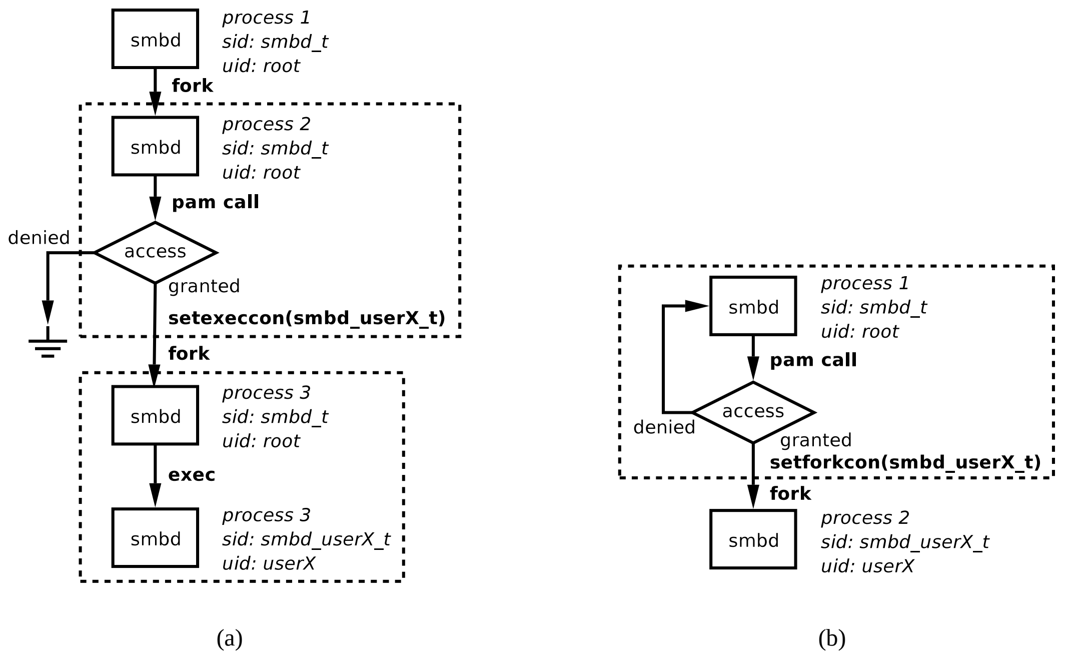


Figure 2: The use of either transitions on exec (*left*) or on fork (*right*) in applications such as *samba* requires thorough modifications in the code of the latter, that are very likely not to be feasible.

lying OS's access control mechanisms are heavily conditioning the architecture of applications. Besides, most of the applications have to be tweaked differently, so one should know the internals of each of them to be able to make the necessary changes, what renders this solution unpractical.

All the problems mentioned above do not apply to applications that execute third party applications, as it is the case of *ssh*, *inetd* and *crond*. For example, *ssh* (see figure 1(a)), every time it receives a connection, forks a child to handle the authentication process, but, unlike *smbd*, once a user has been successfully authenticated, it, once more, forks a new process that will execute (via `execve`) a *shell*. The *shell* is an application different to *ssh* and can thus be naturally `execve`d. The same applies to *crond* and *inetd*, which execute applications completely unrelated to themselves. In the case of *samba* we would be talking about a single application re-`exec`ing itself, which is quite unusual.

No automatic transitions can be defined for *ssh*, since the destination security context depends on the identity of the user that is being authenticated, which, obviously, is not known beforehand. This is the reason why the *pam_selinux* PAM (Pluggable Authentication Modules) module makes use of `setexeccon` to request a domain transition, which is granted if and only if the policy allows *ssh* to transit to that security context using the shell as the entry point.

2.2 *set*uid* calls

The Linux UIDs can be changed at any time using the `set*uid` calls, *providing no control over the inheritance of state or the initialization of the process in the new identity* [3]. The former means that the user to whom the transition is made via any of the `set*uid` calls inherits most of the state from the original user (variables, file descriptors, PID), with the exception, of course, of the new UID. The latter implies that, further to the capability to use the aforementioned functions, there is no control on the way the transition to the new UID is performed. It is simply completely allowed or disallowed. In fact, the root user can adopt any of the UIDs of the system.

Further to this, with the use of some of the *set*uid* functions, a process can transit back to its original UID and then to a different one with no farther controls made by the kernel, which makes the difference between the different process UIDs very thin (from a security point of view). As an example, let's consider a process *P* running as a privileged user *R* that adopts the low privileged identity *U* to perform potentially risky operations. This approach

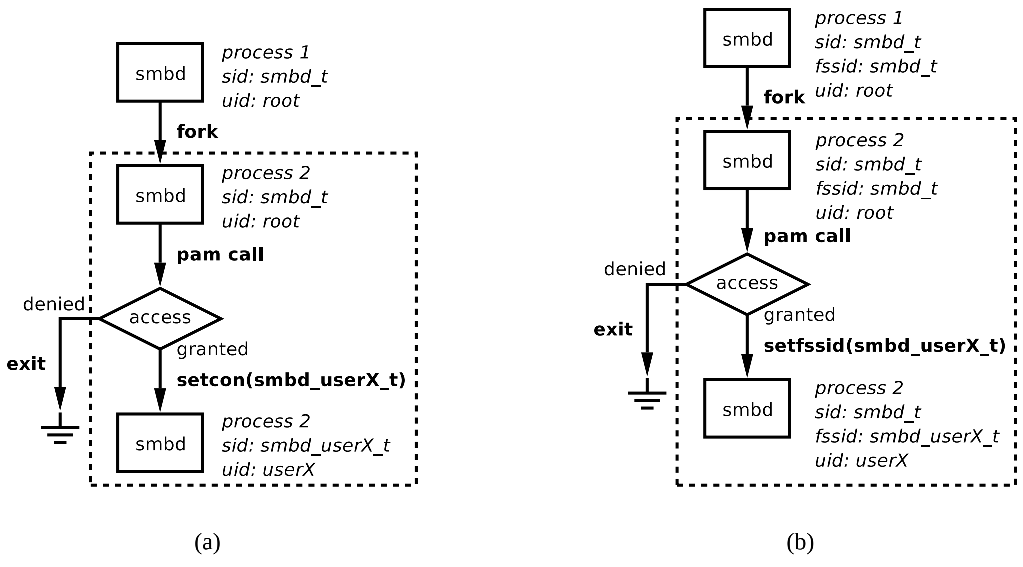


Figure 3: *(left)* `setuid`-like SELinux functionality *(right)* `setfsuid`-like SELinux functionality

may seem safe, but this assumption is flawed. In the event of a buffer overflow bug in the code that runs as user *U*, process *P* could be forced back to *R* for the sake of the execution of malicious code, since *P* is allowed to freely transit back and forth between both identities.

`set*uid` functions are used by, among many others, the smbd daemon (see figure 1(b)), that spawns a child for every user connection. If the client's authentication succeeded, the child transits from the superuser UID to a client specific one, but, from time to time, it needs to revert to root to perform certain tasks. As previously mentioned, an eventual bug in the code could be exploited by the client to become root and execute its own code as a privileged user. This occurs because there's no way for the kernel to tell normal from tampered processes. The damage of such exploits could be minimized if the child process managing the sessions could be run in a less privileged security context, in which access to vital files and resources could be restricted. This is, in fact, the reason why the authors of this paper decided to implement new ways to perform SID transitions (see section 3).

The origin of the unsafeness of `set*uid`-like functions is that the kernel relies all the access control to user space, which is not enough to guarantee the security of a system [4]. User space should not be exclusively responsible for the management of OS objects (such as files storing private keys) whose integrity is critical for the system, since it cannot provide security guarantees without proper OS support. In such cases, the access control should be enforced by the kernel and for that purpose a MAC capable OS is needed.

3 `set*uid`-like functionality for SELinux

The authors decided to implement a set of new functions that provide new methods to perform domain transitions and that are analogical in concept to the standard `set*uid` functions. They were called `setforkcon`, `setcon` and `setfssid`. The first is similar in use to the existing `setexeccon` SELinux function and the other two were implemented after its `set*uid` counterparts.

setforkcon is invoked before the execution of `fork` (just as its model function) and the child is given the security context specified in the call if, and only if, the parent process has been granted the transition to the requested domain (see figure 2(b)).

setcon mimics the standard `setuid` POSIX function in the SELinux world. With it, applications can transit to the desired domain as long as they have been granted the proper permissions (see figure 3(a)).

setfssid, as it name suggests, is analogical to the standard `setfsuid` and allows applications to change its SID for file system accesses, provided that this operation is allowed by the policy (see figure 3(b)).

From now on, the discussion of these new `set*uid`-like functionalities for Linux will be centered on *setforkcon*, except otherwise indicated.

A basic concern aroused before implementing such new functionalities: *would the inheritance of state and the initialization of the process in the new SID be controlled as effectively as it is with the use of* setexeccon *and* execve. The answer to the former is, obviously, no, since, after executing fork, the new process is a copy of the parent, differing only in the PID (as usual) and the SID (that can be changed using the new SELinux library functions). Regarding the initialization in the new SID, it cannot be so precisely controlled, but this was the essential trade-off assumed for their implementation.

The authors think that for applications like Samba, splitting the daemon into a listening server and a session provider that is launched via execve is very similar, from a security point of view, to a new solution using the new SELinux functionalities. Some arguments supporting this thesis follow:

- Before changing the security context functionally identical functions have to be executed (`setexeccon` and `setforkcon` respectively).

- Only if the right permissions (analogical in both cases) were granted in the policy, would the requested transition be performed.

- The initialization of the process is equally controlled with both solutions, since in both cases the children could be spawned only in the set of contexts (one per user) allowed by the security policy. It could be argued that with the use of the new SELinux functions the entry point cannot be specified (since there is no `execve` execution), while using `setexeccon` such a control is possible. But it should be noticed that in this case the entry point would be the current application, not an external one, so there is no point in doing such a distinction.

- There's no need to make the child process execute a program from a type different from the parent's, since both programs belong to the same application and, therefore, it is trusted. Anyways, with the proposed extension the same restrictions could be applied to the child processes.

With the proposed approach only the inheritance of state is weakened, since the basic behavior of fork is not modified and, therefore, the child has access to a copy of the parent's memory at the moment of executing the function. But this is part of the trade-off assumed to ease the adoption of a MAC-capable OS.

It could be thought that the new kernel functionality allows wide and arbitrary changing of context (such as common `set*uid` functions do), but this supposed arbitrariness is constrained to a set of user-specific domains that allow access to user-specific types of files. For this purpose, new process controls have been created (see section 5).

It could be argued that having all the Samba users use the same *samba_staff_t* domain for their I/O would suffice, but, in such scenario, any user could access another user's files in the event of a bug in the code of the child servers.

4 Patches to the Linux Kernel

Though three new functionalities where implemented, this and the following sections will focus on `setforkcon` for the explanation of the methodology used for their implementation. All references to kernel files will be relative to the root of the source tree.

To allow *domain transitions on fork*, the kernel needs to be modified as follows:

- `fork` must be extended so that it can change its child's context (see section 4.1).

- User space applications must be provided with a proper interface to the new SELinux functionality (see section 4.2).

- The adequate LSM hooks must be modified to implement the new control functionalities (see sections 4.1 and 4.3).

- New permissions have to be created so that the use of `setforkcon` can be controlled (see section 4.4).

4.1 Extending fork

The existing `fork` functionality was extended in a way that full compatibility with the original version is preserved. Thus, on the one hand, existing applications can be used without requiring any change and, on the other hand, it can be used by those willing to take advantage of its new capabilities.

Due to the modular design of LSM it was not necessary to modify the kernel/fork.c file. Since fork makes use of the LSM hook security_task_alloc to set the security attributes of the new process, only the SELinux implementation of that function must be altered to provide the desired features. The SELinux module implements this hook in the file security/selinux/hooks.c as the function selinux_task_alloc_security. Within this function the requested transition is checked and, if the security server grants permission, the transition is actually performed.

4.2 Interface to the new SELinux functionalities

As mentioned on previous sections, SELinux only allows transitions on execve, but this transitions can be performed automatically or under request. In the former case, the transition is defined in the policy and it is automatically attempted whenever execve is executed, transparently to the application. In the latter case, before forking a child, the parent process executes a SELinux library function (see section 6 for further details) to set the transition to be attempted. Once the child has been spawned, if it executes a execve function, the transition defined by the parent will be tried and, if it is allowed by the policy, performed.

Analogically to the on-demand execve transition, applications were provided with a method to request a determined transition on fork, which is achieved by adding a new file to the /proc/*/attr/ directories. The files in this process-specific directories serve as the interface to several SELinux-related attributes. Usually they are read-only files, but in the case of /proc/*/attr/exec, it can be used to set the desired transition. To create the new file fork, fs/proc/base.c in the Linux kernel tree must be patched. The main modifications are summarized below:

- The enumeration pid_directory_inos must be added two fields per file to be created. In the case of fork, and following the example of the existing entries, PROC_TGID_ATTR_FORK and PROC_TID_ATTR_FORK were added.

- The array tgid_attr_stuff was added a new pid_entry struct, analogical to that corresponding to the exec file.

- Similarly, the array tid_attr_stuff was added the struct corresponding to fork.

- The function proc_pident_lookup must also be modified so that, whenever a access is attempted to the fork file, an adequate function is called to provide the answer. To follow the SELinux's procedure, the struct proc_pid_attr_operations is used for this purpose (a brief explanation on this topic is provided in 4.3). Thus two new cases must be added somewhere in the sequence of cases that precedes this function call. As expected, PROC_TID_ATTR_FORK and PROC_TGID_ATTR_FORK are the cases to add.

4.3 proc LSM hooks

As its name suggests, the aforementioned proc_pid_attr_operations is the struct that keeps the pointers to the functions that deal with read and write operations on the files in the attr directory. This functions are proc_pid_attr_read and proc_pid_attr_write, respectively. Both functions have LSM hooks (security_getprocattr and security_setprocattr) that are implemented by SELinux in the file security/selinux/hooks.c (in the functions selinux_getprocattr and selinux_setprocattr).

These two functions set and read the security attributes of the running process if adequate permissions have been granted. Every process stores its SELinux security attributes in a task_security_struct struct that is defined in security/selinux/include/objsec.h. For the sake of setforkcon a new field fork_sid was added. In fact, this is the variable read by the selinux_task_alloc_security hook (revise section 4.1) to set the security context of the child when fork is executed.

4.4 Creation of new permissions

If the SELinux module is loaded into the kernel and enabled (i.e. in enforcing mode), when a process tries to perform an operation on an object (e.g. file, socket, IPC object, etc) the security server bases its access decision on the security context of the process, the security context of the object that it is trying to access, the class of that object (file, socket, process, etc) and, of course, the operation itself. The set of operations that can be performed on an object depends on its class. For example, for a process object operations such as *fork*, *transition* and *setexec*; and for a file object, among many others, there are *ioctl* and *read* operations.

SELinux is able to control many of the operations that can be performed on kernel objects, but, for this purpose, it needs to assign a unique label to the object classes meaningful to be the policy and, for every class, the operations to be controlled must also be defined. This is achieved by means of the file `flask/access_vectors` under the policy definition directory[2], file in which all the classes, and the operations on them that are controlled, are defined.

Since we are defining new operations on processes (i.e. `setforkcon`, `setcon` and `setfssid`), proper entries must be added to the aforementioned file under the process class definition (see section 5).

The policy is defined, compiled and loaded into the kernel from user space, but this policy should only reference object classes and operations that the kernel knows about. Otherwise, the policy is meaningless to the kernel. This is why any change in the file `access_vectors` (or any other data file under the `flask` directory) should be immediately reflected in the kernel (see section 5). Of course, for the changes to take effect the kernel must be recompiled and the system rebooted.

5 SELinux Policy modifications

Since new operations were implemented into the kernel and we want to make SELinux aware of them to be able to control them. As previously mentioned (see section 4.4), the SELinux policy is defined outside the kernel's source tree and the definitions under the `flask` directory must be synchronized with the kernel.

In the case of `setforkcon`, first the `access_vectors` file was modified as shown in listing 1. Afterward, the Makefile under the `flask` directory should be used to generate the header files needed by the kernel and that must be copied into the `security/selinux/include` kernel's directory.

Finally, for the user-space to be aware of the object classes and operations, the same header files should also be copied to the `/usr/include/selinux` directory. This is not always necessary, though, since applications hardly ever need to know about this information.

[2]The SELinux policy directory is usually located somewhere under `/etc` and is, thus, completely unrelated to the kernel's source tree. In the case of the Debian the directory is `/etc/selinux`.

Listing 1: *selinux/flask/access_vectors* excerpt

```
class process
{
  fork
  transition
  .....
  fork_transition
  fs_fork_transition
  .....
  setexec
  setfork
  setfssid
  setfscreate
  .....
}
```

6 Kernel/User space interface to the new kernel functions

The access to the new kernel functionalities has been largely inspired by the current implementation of the library function *setexeccon*, which interfaces with the kernel by means of the */proc/*/attr/exec* files (see line 11 of code listing 2). A process could access this file directly but the use of the wrapper is a cleaner and, in case the name of the file changed or a completely new interface to the kernel (such as a system call) was offered, only the library would have to be changed, not all the applications.

Thus, three new library functions where added: `setforkcon`, `setcon` and `setfssid`.

7 Practical case: Samba

Although throughout this paper `setforkcon` was used to explain the implementation of `set*uid` like functionality onto the Linux kernel, to secure Samba the `setcon` function was used (`setfssid` was also be a good choice).

Listing 2: *setforkcon.c* excerpt

```
fd = open("/proc/self/attr/fork",
    O_RDWR);
if (fd < 0)
  return -1;
if (context)
  ret = write(fd, context, strlen(
     context)+1);
else
  ret = write(fd, NULL, 0);
close(fd);
```

As it was previously mentioned the objective was to run the Samba user sessions in a user-specific domain. For that, after successfully authenticating a user, the `smbd` daemon should call `setcon` indicating the security context corresponding to the authenticated user. If the policy allows the transition to the user-specific domain will be performed.

But, since Samba requires powerful permissions to run, the previous is not enough. If `smbd` adopted the basic `user_t` after a user's successful authentication, it would die away right after that, due to the `user_t`'s restrictive policy, that is not enough to carry out the tasks required by a Samba session. Further to this, the aforementioned user-specific domains should not be the ones granted to the user on standard login sessions, since for the latter is usually enough with a domain of the `user_t`'s characteristics.

As example, let's consider a *userx* that successfully establishes a Samba session with the modified `smbd` daemon. The UID of the `smbd` child process would be `userx` and its security context `userx:smbd_r:smbd_userx_t`:

```
root@SELINUX:~> (ps ax -Z;pstree)|
    grep smbd|grep -v grep
  498 system_u:system_r:smbd_t   /usr
     /sbin/smbd -D
 5684 userx:smbd_r:smbd_userx_t  /usr
    /sbin/smbd -D
     |-smbd---smbd
```

The way in which Samba was patched and the policy changes related to the user-specific security contexts is not covered in this paper.

7.1 Installation

In short, to install a security enhanced Samba the following steps were followed:

- Modify the file `access_vectors` under the SELinux policy directory tree (see section 4.4).
- Out of the previous file generate and install the necessary header files for the Linux kernel and the user-space SELinux libraries (see section 5).
- Patch, recompile and install the Linux kernel (see section 4).
- Patch, recompile and install Samba.
- Define a suitable policy for Samba.
- Recompile the binary policy.
- Reboot the system.
- Load the new policy (this is usually done automatically after rebooting).

8 Conclusions

The authors' first impression towards SELinux was great, because SELinux along with the LSM was finally providing Linux with truly advanced and flexible access control mechanisms. However, trying to set up a production environment, what appeared to be the SELinux shortcoming showed up: SELinux only allows security transitions on `execve`.

But, in fact, the latter was one of the original design decisions and could be even be considered a good security feature, since it does impose harder conditions for security domain transitions. But, on the other hand, it does not contemplate the needs of applications that, while managing user sessions, do not rely their management on a different application. This is the case, among many others of *Samba*, which due to the aforementioned imposition can hardly benefit from the existing SELinux implementation.

For this reason the authors decided to extend the SELinux functionalities, for which purpose both its kernel and user-space parts had to be modified. This task was not as hard as expected, since LSM as well as SELinux were originally designed so that they are almost non-intrusive with respect to the rest of the kernel's code. Thus, the parts that need to be modified are quite easily located.

Regarding the modification of the applications, it is greatly simplified with the adoption of the new functionalities. Usually, only the part of the application dealing with authentication has to be modified, so that, after successfully login, one of the new functions is executed (thus running the session in the adequate security context).

A task that remains pretty painstaking is the definition of an appropriate policy, although, to ease this task, suitable new macros should be created.

Finally, the authors consider that with the new functions, the somewhat unbalanced security vs usability trade-off of the original SELinux is more equilibrated.

9 Future work

Due to the Windows client's lack of support for SELinux, it can neither handle nor understand SIDs, which forces the current implementation of Security Enhanced Samba to rely exclusively on the Samba user identity to perform access control. Further to this, there is not an underlying infrastructure providing network-wide SIDs. Research on this topics is currently being undergone by the authors of this article and any positive results will, of course, be shared with the whole Linux community under the terms and conditions of GPL.

10 Acknowledgments

Thanks to the EU-Japan Centre for Industrial Cooperation, NTT Data Corporation and the University of Vigo for supporting my research work in Japan. In other order of matters, thanks to all the people involved in the Linux, LSM and SELinux projects.

11 Acronyms

OS Operating System

SELinux Security Enhanced Linux

UID User Identifier

MAC Mandatory Access Control
The need for a MAC mechanism arises when the security policy of a system dictates that protection decisions must not be decided by the object owner and, besides this, the system must enforce the protection decisions (over the wishes or intentions of the object owner).

SID Security Identifier
The SID is a local, non-persistent integer that is mapped by the security server to a security context.

LSM Linux Security Module
Infrastructure that provides the Linux kernel with the ability to load security modules that perform additional access control routines.

PAM Pluggable Authentication Modules
PAM is a suite of shared libraries that enable the local system administrator to choose how applications authenticate users. It is even possible to change the authentication mechanisms of PAM-aware applications, without rewriting or recompiling them.

References

[1] Faye Coker. Getting started with se linux howto: the new se linux, March 2004. `http://www.lurking-grue.org/GettingStartedWithNewSELinuxHOWTO.pdf`.

[2] Faye Coker. Writing se linux policy howto, March 2004. `http://www.lurking-grue.org/WritingSELinuxPolicyHOWTO.pdf`.

[3] Stephen Smalley. Configuring the selinux policy, January 2003. `http://www.nsa.gov/selinux/papers/policy2/t1.html`.

[4] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computer environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998. `http://www.nsa.gov/selinux/papers/inevit-abs.cfm`.

[5] Department of Defense. TCSEC (trusted computer system evaluation criteria), December 1985. DoD 5200.28-STD, `http://csrc.ncsl.nist.gov/secpubs/rainbow/std001.txt`.

[6] Serge E. Hallyn. Domain and type enforcement for linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[7] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Proceedings of the Ottawa Linux Symposium*, 2002. `http://lsm.immunix.org/docs/lsm-ols-2002/html/`.

[8] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module, May 2002. `http://www.nsa.gov/selinux/papers/module/t1.html`.

[9] Rule set based access control (rsbac) for linux. `http://www.rsbac.org/`.