RootAsRole: a security module to manage the administrative privileges for Linux

Ahmad Samer Wazan
College of Technological Innovation
Zayed University
Abu Dhabi, UAE
Email: ahmad.wazan@zu.ac.ae

Eddie Billoir
IRIT Laboratory
Paul Sabatier University
Toulouse, France
Email: eddie.billoir@irit.fr

David W Chadwick
Crossword Cybersecurityg
Canterbury, Kent
Email: david.chadwick@crosswordcybersecurity.com

Romain Laborde
IRIT Laboratory
Paul Sabatier University
Toulouse, France
Email: laborde@irit.fr

Mustafa Kaiiali
School of Computer Science and Informatics
De Montfort University
Leicestershire, UK
Email: mustafa_kaiiali@ieee.org

Remi Venant
Computer Science Department
University of Le Mans
Le Mans, France
Email: remi.venant@univ-lemans.fr

Liza Ahmad
College of Technological Innovation
Zayed University
Abu Dhabi, UAE
Email: Liza.ahmad@zu.ac.ae

Abdelmalek Benzekri
IRIT Laboratory
Paul Sabatier University
Toulouse, France
Email: benzekri@irit.fr

***Abstract*—Today, Linux users use *sudo/su* commands to attribute Linux's administrative privileges to their programs. These commands always give the whole list of administrative privileges to Linux programs unless there are pre-installed default policies defined by Linux Security Modules (LSM). LSM requires users to inject the needed privileges into the memory of the process and to declare the needed privileges in an LSM policy. This approach can work for users with good knowledge of the syntax of LSM policies. However, adding or editing an existing policy is very time-consuming because LSM requires adding a complete list of traditional permissions and administrative privileges. Therefore, we propose a new Linux module called *RootAsRole* dedicated to managing administrative privileges. *RootAsRole* is not proposed to replace LSM but to be used as a complementary module to manage Linux administrative privileges. *RootAsRole* allows Linux administrators to define a set of roles that contain the administrative privileges and restrict their usage to a set of users/groups and programs. Finally, we conduct an empirical performance study to compare *RootAsRole* tools with *sudo/su* commands to show that the overhead added by our module remains acceptable.**

***Index Terms*—sudo/su, Linux capabilities, privilege escalation, access control.**

1. Introduction

Administering an OS includes many tasks, such as managing the OS users, file system, security policy, processes, system clock, etc. Historically, Linux administration was based on the existence of one powerful user, called superuser or root, whose id value is 0. The initial administration model was straightforward because any user can manage any resource on the system as long as the effective user ID (*euid*) of the process run by the user is equal to zero. Indeed, every process on Linux systems has four types of group and user IDs, the most important ones are Real ID (*ruid*,*rgid*) and Effective ID (*euid*,*egid*). Typically, real ID is used to show to which user or group the process belongs, while Effective ID determines the permissions granted to a process. According to the initial administration model, any process whose *euid* or *egid* is 0 can achieve all the administrative tasks on Linux. This type of process is referred to as a privileged process [11]. Consequently, any regular user who wants to achieve an administrative task on Linux must change the process's *euid* or *egid* to 0.

This administrative model can lead to a privilege escalation attack when the executed program has an exploitable vulnerability or when the executed program has some arguments that allow malicious users to execute arbitrary commands on the system. For example, in 2019, a vulnerability was discovered in the *sudo* program (setuid bit set) that

allows any non-privileged user to become root [21]. Another example is an administrator who adds to the *sudo* configuration file the possibility for a user to run the command *find*. This command could give the user root access to the whole system if the administrator was not aware of the existence of the *exec* argument in the *find* command. The *exec* argument allows the user to execute arbitrary commands on the system (see Figure 1). In addition, the simple administration model can also cause disastrous problems in the system when users commit some fatal error such as *"rm -rf /"* which allows the user to remove the whole files system[1] (see Figure 2).

```
tomas@osboxes:~$ sudo find /home/tomas -exec sh -c "whoami" \;
root
```

Figure 1. Privilege escalation with *find* command.

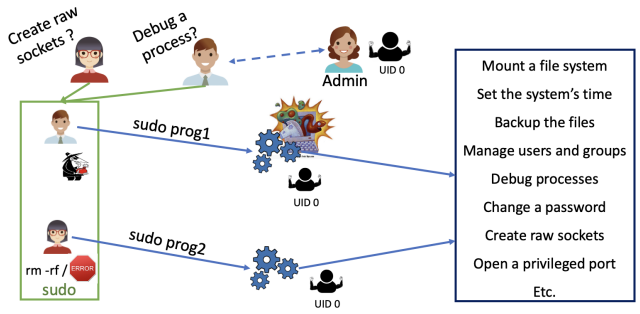

Figure 2. *sudo* command.

As a consequence, POSIX draft 1003.1e has been proposed to create special permissions called Linux capabilities[2]. Linux capabilities are a set of administrative privileges that allow their owner to execute different administrative tasks (the terms Linux capabilities and administrative privileges are interchangeably used in this paper). Although the POSIX draft has been withdrawn, it has been integrated into the kernel of Linux since 1998. The power of the super user is divided into a list of sub-powers that can be distributed separately to processes by giving them only the administrative privileges they need [1] (see Figure 3).

For different reasons, Linux capabilities have not been widely used. The first problem comes from the use of extended attributes to store the capabilities of executable files (problem 1). Secondly, Linux administrators do not have a tool that allows them to distribute capabilities to Linux users in a fine-grained manner (problem 2). Fine-grained privilege distribution should give an administrator the ability to decide: (1) which capabilities to give to which users or groups, (2) with which programs, and (3) on which resources users can use the granted privileges. Finally, Linux does not provide a tool that permits Linux users to know the capabilities that a program needs in order to run successfully (problem 3).

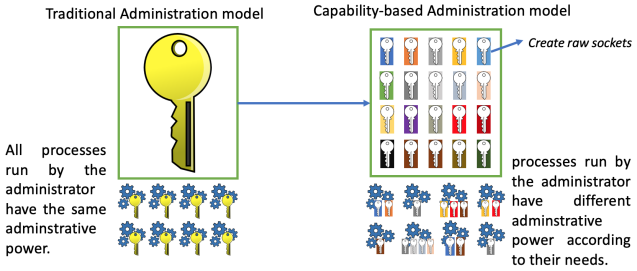

Figure 3. Traditional-based processes vs. capability-based processes.

In 2015, the kernel of Linux was modified in order to address the technical problems related to the storing of capabilities in the extended attributes of executable files (i.e. problem 1). This new feature added to the kernel makes it possible to extend the capability module by adding on top of it a policy management tool that can be configured by the administrators of Linux systems [31]. The objective of this paper is to allow administrators to restrict the use of Linux capabilities in their systems by resolving problems 2 and 3. Concretely, we are providing a module called *RootAsRole* that gives system administrators the possibility of finely controlling the distribution of administrative privileges to Linux users. We have implemented a role-based approach where each role maps to a set of capabilities that can be granted to users or groups of users. Linux users use our *sr* command to assume these roles and execute their privileged applications. In addition, we provide a command called *capable* which allows Linux users to know the administrative privileges requested by Linux users. Finally, we provide the commands *addrole*, *editrole* and *deleterole* that allow a Linux administrator to edit the policy of *RootAsRole*.

Linux Security Modules (LSM) can be used to restrict administrative privileges by requiring (1) the injection of the needed privileges into the memory of processes and (2) declaring them in the LSM policies. Practically, Linux users always use *sudo/su* commands to inject the administrative privileges and depend on the program's author to define an LSM policy. *sudo/su* commands permit users to inject the full list of administrative privileges into the memory of processes because Linux kernels come with an emulation mode that allows any process whose *euid* is 0 to have the full list of Linux capabilities. Whether a program has an LSM policy or not, Linux users can not notice when a process is confined or not as they are always using *sudo/su* commands. When a process has no LSM policy, Linux administrators need to write LSM policies that have to contain rules for the traditional permissions and the administrative privileges. *RootAsRole* allows Linux administrators to only edit rules for administrative privileges while allowing them to still use the LSM if they need to manage the traditional and the administrative rules.

This paper expands our previous work [25] by:

1. In 2006, GNU *rm* added the option -no-preserve-root to prevent this problem

2. The capability term here should not be confused with the capability term used in access control literature which refers to a token given by the kernel to a process to access an object (e.g. file descriptor)

1) Providing a clear explanation about Linux capabilities and its history, which is difficult to understand due to its complexity,
2) Discussing the rationale of the *sr* command implementation,
3) Adding more motivation scenarios to prove the applicability of our proposal,
4) Defining an additional command called *capable* that allows users to know which capabilities are being requested by programs (i.e. problem 3). We have used eBPF which allows us to intercept the syscalls of the Linux capability module and return the requested capability.
5) Adding a set of tools that allow an administrator to add/update/delete roles without needing to edit the configuration file manually. This can reduce the configuration errors that can be accidentally made by administrators.
6) Providing a detailed analysis of the inefficiency of LSM in managing the Linux administrative privileges,
7) Discussing the complexity overhead added by our tools when they are compared with the *sudo* command.

The paper is organised as follows. Section 2 describes Linux capabilities, showing how they are calculated by the Linux kernel and what limited tools are available to distribute them to Linux programs. In section 3, we review related work, whilst section 4 introduces the *RootAsRole* module and shows its advantages by presenting two motivating scenarios. We describe the implementation in section 5 and in section 6 we conclude with the limitations of our proposal and our future work.

2. Linux Capabilities

In 2000, starting with Kernel 2.2, Linux divided the traditional power of the superuser into smaller distinct units called capabilities [12]. There are currently 38 capabilities implemented in the kernel of Linux. In this system, the notion of a privileged process changes to represent any process that has one or more of the 38 capabilities in its credentials and is no longer a process whose *euid* or *egid* is equal to zero. The root user is now considered as any other regular user. However, Linux still provides an emulation mode that allows any process whose *euid* or *egid* is equal to zero to automatically have the full list of available capabilities in its credentials and thereby become a privileged process.

All Linux capabilities start with the keyword *cap*, and each one of them allows a different set of administrative tasks. For example, any process that has the capability *cap_net_admin* can modify the network interface configuration, administer the IP firewall, and modify the routing tables as well as many other network-related activities. The capability *cap_dac_override* allows the process to bypass file and directory permissions. Similarly, *cap_mac_override* allows the Mandatory Access Control (MAC) rules to be overridden. A complete explanation of the Linux capabilities can be found on Linux manual page - capabilities(7) [12].

In response to the presentation of the *SELinux* project by the United States NSA in 2001, Linux Torvalds defined a general security framework, called Linux Security Modules (LSM), that provides a set of security hooks in the kernel objects. These hooks can be implemented by different security modules whose behaviours are controlled by a policy. One of the most known modules are *SELinux*[3], *AppArmor*[4], and *grsecurity* [5]. In addition, Linus Torvalds asked to migrate the logic of Linux capabilities into the LSM security framework [10]. LSM framework allows to implement additional access control models such as Mandatory Access Control (MAC) and Role Based Access Control (RBAC) models. The capability's logic is implemented inside the LSM framework [10] (Figure 4). However, unlike other LSM such as *SELinux* and *AppArmor*, the capability module cannot be removed from the LSM and has no policy store to control its behaviour.

The LSM hooks are inserted inside syscalls, which allow different security models to be integrated into the Linux kernel. Whenever a privilege is needed during the execution of a syscall (e.g. *fork()*), a *capable()* or *ns_capable()* function is called to check whether the calling process possesses the requested privilege or not. These functions call another function, called *cap_capable()* that is implemented inside the capability module of the LSM[6].

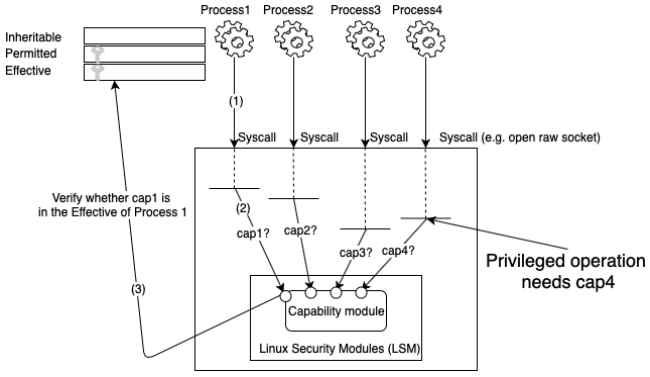

Figure 4. How capability's check works.

At the process level, when we look at the *cred* structure (Listing 1) in the Linux kernel [28], which defines the security context of a task, we find 5 sets (fields or bitmasks) used to store the Linux capabilities: Effective, Permitted, Inheritable, bset (or Bounding) and Ambient. Each of these sets has a different purpose.

The file /proc/[PID]/status lists the current values of the five sets for the process with the given PID. For any root process, the values of Permitted, Effective and Bounding capability sets are totally filled, whilst the Inheritable and Ambient sets are totally empty (Figure 5). Since the reserved

3. https://github.com/SELinuxProject
4. https://apparmor.net
5. https://grsecurity.net
6. This behaviour changes when another LSM is installed such as *AppArmor* or *SELinux*. The behaviour is explained in section 3

```
struct cred {
....
kernel_cap_t   cap_inheritable;
/* caps our children can inherit */
kernel_cap_t   cap_permitted;
/* caps we're permitted */
kernel_cap_t   cap_effective;
/* caps we can actually use */
kernel_cap_t   cap_bset;
/* capability bounding set */
kernel_cap_t   cap_ambient;
/* Ambient capability set */
.....
}
```

Listing 1. Cred sturcture [28]

storage size for each capability set is 64 bits, and Linux has only defined 38 capabilities, this explains why the top 26 bits of each set are empty.

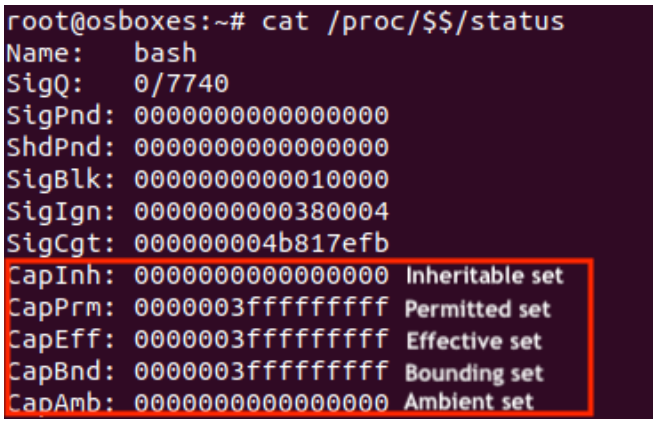

Figure 5. Values of a shell run by root (cat /proc/$$/status).

Effective contains the set of capabilities that are currently used by the process. *cap_capable()* reads this set in order to verify whether a process is allowed to achieve a privileged task or not. Permitted contains the set of capabilities that a process can use, which is a superset of the Effective set. A process can drop a capability from its Effective and Permitted sets. Dropping a capability from the Effective set means that the process wants to temporarily disable the concerned capability, but removing a capability from the Permitted set means that the process permanently loses the capability. The Inheritable set is used by a process that wants to grant some capabilities to another process that results from an *exec()* call to its binary file. If a process has the *cap_setpcap* capability in its Permitted, this gives it the possibility to add additional capabilities to its Inheritable set. *cap_bset* is the capability bounding set of a process. This set is used to limit the capabilities that it may pass on to a new process obtained from an *exec()* call to its executable file. Without this set, attackers, who succeed in modifying the extended attributes of an executable file, could run processes that have the full set of root privileges. The *cap_bset* set is also considered to be a superset of the Inheritable set. The Ambient set is a relatively new addition to the Linux Kernel, which has been added to resolve the problems arising from the use of extended attributes to store the capabilities of executable files (see section 2.2).

Executable files only have the ability to store three sets of capabilities: Inheritable, Permitted and Effective. These sets are used to grant privileges to the process resulting from an *exec()* call of the binary file, by masking them against the capability sets of the calling process as described in the next section.

2.1. Capabilities calculation

After an *exec()* call of program (F) by the calling process (P), the Linux kernel applies the following rules[7] to calculate the capabilities of the new process (P'):

$$
\begin{aligned}
P'_{Permitted} &= (P_{Inheritable} \& F_{Inheritable}) | (F_{Permitted} \& P_{cap\_bset}) \\
P'_{Effective} &= F_{Effective} ? P'_{Permitted} : 0 \\
P'_{Inheritable} &= P_{Inheritable}
\end{aligned}
\tag{1}
$$

Where:
& is bitwise AND,
| is logical OR,
$z = x?y : 0$ means if x has a value or true; z will take the value of y otherwise z will be zero.

These rules show that the Permitted set can be filled either through inheritance by masking the Inheritable set of the calling process ($P_{Inheritable}$) against the Inheritable set of its binary file ($F_{Inheritable}$), or directly from the Permitted set of its binary file ($F_{Permitted}$) after having masked it against the *cap_bset* (or Bounding) set of the calling process P. The Effective set of P' will have an exact copy of its Permitted set if the file's Effective set is enabled, otherwise it will be empty. Finally, its Inheritable set will have the same values as the Inheritable set of the calling process P.

For example, say an administrator wants to give a set of users the possibility to use *tcpdump*. In terms of capabilities, *tcpdump* requires the privileges *cap_net_raw* and *cap_net_admin*. An administrator, providing it owns the privilege cap_setfcap, can use the command *setcap* to grant these privileges to the *tcpdump* file, as follows:

*setcap cap_net_raw, cap_net_admin=**ep** /usr/sbin/tcpdump*

This command stores the *cap_net_raw* and *cap_net_admin* in the Effective and Permitted sets of the *tcpdump* binary file (**ep** refers to the Effective and Permitted sets), assuming the system administrator's Bounding set contains these capabilities. Whenever a Linux user runs *tcpdump* from its shell, the shell makes an *exec()* call to the *tcpdump* file. In this case, the Linux kernel

7. These rules were prior to the addition of the Ambient set in 2015, and they are not applied when a process creates another process using the *fork()* call. The child process resulting from a *fork()* call always has the same capabilities as its parent.

applies the calculation rules to determine the capabilities of the new process P'. P' will only have *cap_net_raw* and *cap_net_admin* in its Permitted set if the user's process has them in its Bounding set. The Permitted set of P' will be copied into its Effective set because the Effective set of the *tcpdump* file has been set. Thus any user that has the Discretionary Access Control (DAC) permission to execute *tcpdump* can run it without the need to have a root account.

If the administrators desire to restrict the execution of *tcpdump* to certain users, they should create a group that is composed of the users allowed to execute *tcpdump* and then change the DAC permission of *tcpdump* to restrict the execution to the created group.

Another option an administrator may follow is to use the *pam_cap* module. This allows the *cap_net_raw* and *cap_net_admin* capabilities to be given to users, once they are logged in to the system. The basic idea of *pam_cap.so* is to distribute Linux capabilities to users by applying the inheritance rule. The module *pam_cap.so* comes with a configuration file (/etc/security/capability.conf) that allows the Linux administrators to define the list of capabilities that they want to grant to users. The first processes that run (i.e. *init* and *login*) during the booting of a Linux system are privileged processes that own all the capabilities in their Permitted and Effective sets. As a consequence, they can fill the Inheritable set of their descendant processes like a user's shell. The *login* process loads the *pam_cap* module that injects the privileges in the Inheritable set of its own process and then it creates the shell process (via *fork()* and *exec()* calls). As a result, the capabilities defined by the administrator in the "capability.conf" file will be stored in the Inheritable set of a user's shell. However, having the capabilities in the Inheritable set would not allow to the user to run *tcpdump*; the administrator must also inject the same capabilities into the Inheritable and Effective sets of the *tcpdump* file, using the command:

*setcap cap_net_raw, cap_net_admin=**ie** /usr/sbin/tcpdump*

where **ie** refers to the Inheritable and Effective sets. Only users that have the same capabilities in their Inheritable set of their shell will be able to run the *tcpdump* command.

2.2. Problems with the Linux Capability File-based Approach

Linux stores the file capabilities in the extended attributes (*xattrs*) of binary files. This has been the cause of several different problems that has limited the use of the Linux capability model. In particular:

- Some basic Linux commands do not use the extended attributes correctly. For example, the *mv* command preserves the extended attributes by default. However, the administrator does not get a warning message when moving files onto a file system that does not support the extended attributes, so they are lost. On the other hand, the *cp* command does not copy the extended attributes by default; the administrator must add the option *--preserve=xattr*. Other issues have been reported about archiving and backup tools that do not properly take care of the extended attributes [2];
- Executing privileged scripts in a secure way is not possible. Linux administrators have to inject the root privileges into the interpreter program binary and not into the scripts. In this way, all scripts run by the interpreter will gain the privileges of their interpreter whereas the administrator may wish to give different privileges to different scripts;
- The *xattrs* are often lost when Linux packages are updated. This is a huge problem when the number of binaries containing capabilities is relatively big;
- The Linux kernel does not take into account the configuration of the *LD_PRELOAD* variable when capabilities are stored in binaries. *LD_PRELOAD* is an environment variable that contains the list of user-specified libraries that are dynamically loaded before all other shared libraries [3]. Typically, this feature can be used to intercept the standard system calls such as *malloc()*, *open()*, *close()*, etc.

It should be noted that it is not possible to completely remove the need to store capabilities in the *xattrs* of executable files but a mechanism has been introduced to limit their use. This is the idea of the Ambient set (see section 5.1). However, today there are no tools that permit the distribution of capabilities to Linux users who wish to use the Ambient set. Administrators have no choice but to use the *pam_cap* module and the extended attributes of executable files to distribute capabilities to their users. We provide an alternative to *pam_cap* that does not necessitate storing the capabilities in the extended attributes of executable files.

3. Related Works

LSM (e.g. *SELinux* or *AppArmor*) are today the main tools that allow restricting the usage of administrative privileges in Linux. The LSM' logic to manage administrative privileges is different from *setcap*, *pam_cap* and our module *RootAsRole* because LSM are designed to check first whether the needed privilege is owned by the process and then check whether the needed privilege is declared in an LSM's policy. Figure 6 illustrates the current decision algorithm implemented by the Linux kernel for LSM. We can notice two different paths: the first one is to check traditional permissions (i.e. whether the process owns traditional permission- read/write/execute) as per DAC model rules, and the second one is to check the administrative privileges (i.e. Linux capabilities).

For example, the hook function *apparmor_capable()* (Listing 2) implemented in *AppArmor* checks first whether the tested capability (*cap*) is present in the *cred* data structure of the process (using *cap_capable()*). It then checks whether the process has a pre-defined profile (i.e. confined) and whether the tested capability *cap* is present in the profile using the function *aa_capable()*. If the process is not

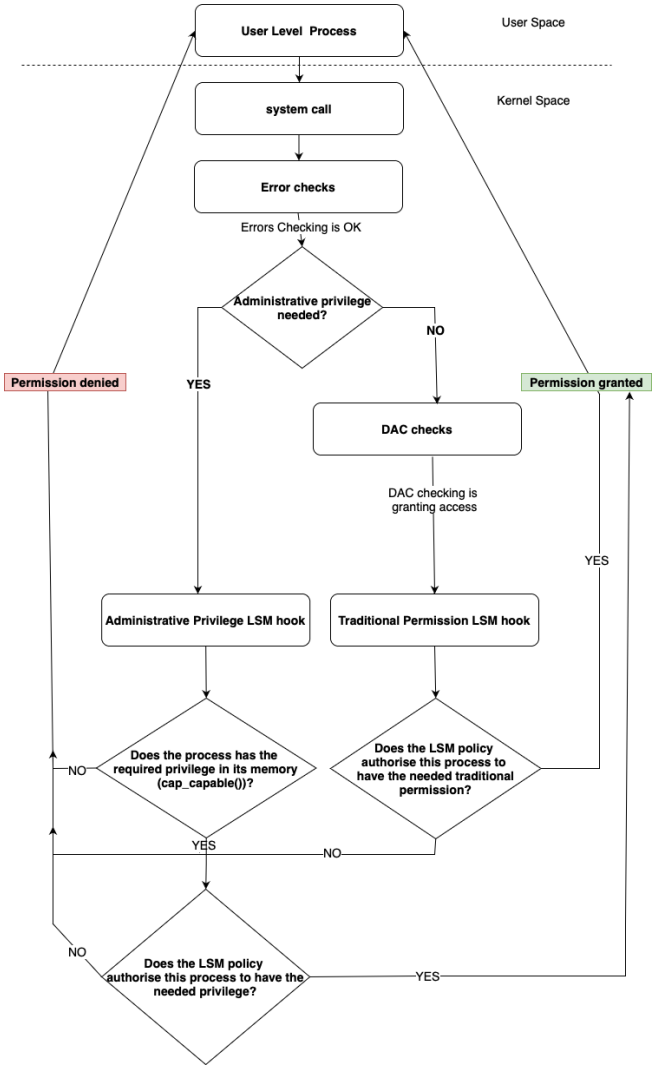

Figure 6. Current access control decision algorithm implemented by Linux

confined then only *cap_capable()* is checked and its result is returned.

Consequently, Linux users have to inject the needed capability into the memory of the process and also declare the needed capability in the policy of the LSM. Typically, Linux users use *sudo/su* commands to inject the whole list of capabilities into the memory of the process and then mask them with the list of capabilities declared in the policy of the LSM. This approach can work well if there is a complete list of pre-installed policies for Linux processes and programs used by users or when users can easily add an LSM policy. First, LSM do not provide by default the whole list of Linux programs and processes. For example, the default policy for *SELinux* is *Targeted policy*. This policy confines only a subset of the Linux processes that are considered risky.

Similarly, the pre-installed profiles of *AppArmor* do not cover all Linux programs. For example, there is no pre-installed *AppArmor* profile for the *find* command on *Ubuntu*; a user has to define a profile for this command

```
static int apparmor_capable(const struct cred *
    cred, struct user_namespace *ns,
          int cap, int audit)
{
  struct aa_profile *profile;
/*cap_capable returns 0 on success,else -EPERM */
  int error = cap_capable(cred, ns, cap, audit);
  if (!error) {
    profile = aa_cred_profile(cred);
    if (!unconfined(profile))
      error = aa_capable(current, profile, cap,
    audit);
  }
  return error;
}
```

Listing 2. Code source of the function apparmor_capable() [29]

to protect himself from the escalation attack explained in the introduction section. Second, adding an LSM policy is complex because users have to add a complete list of rules that cover administrative privileges and traditional permissions. The All-or-nothing approach can encourage users to run their programs without restriction because defining the whole list of rules is a very time-consuming process. For example, to restrict the Server web developed in Python using *AppArmor*, we have to add more than 100 rules for traditional permissions (added by a set of include statements or directly in the policy (see Listing 3) in order to add only one Linux capability in the *AppArmor* profile. *RootAsRole* proposes a complementary approach to LSM as it allows the management of administrative privileges exclusively. So when an administrator defines the needed privileges in *RootAsRole*'s configuration file, only the needed capabilities will be injected into the memory of the process, and the function *apparmor_capable()* will continue to work because *cap_capable()* function will return 0 (i.e., success) as the needed capabilities are present in the memory of the process. Our module allows then Linux users to restrict the usage of administrative privileges without needing to define access rules for traditional permissions. Linux administrators can still use the LSM to define rules for traditional permissions and administrative privileges if needed.

In addition, LSM can have additional issues that can be considered implementation-dependent. *AppArmor* is considered easier than *SELinux* because it supports only the Type Enforcement (TE) access control mechanism, and uses pathname enforcement instead of labels. In addition, *AppArmor* provides a learning tool for users to facilitate the development of profiles. Defining a profile can be completed using the command *aa-genprof* which allows users to generate or update profiles. When using this command, the user is required to specify the program to profile. The user then runs the program at the same time and the tool then asks the user/administrator to approve the addition of each generated rule to the profile using various options such as Allow, Deny, Abort, etc. *AppArmor* also provides the users with two methods of profiling: Stand-alone profiling and Systemic profiling. While standalone profiling using *aa-genprof* com-

```
abi <abi/3.0>,
include <tunables/global>

/home/kali/server3.py {
  include <abstractions/apache2-common>
  include <abstractions/base>
  include <abstractions/python>

  /home/*/ r,
  /home/kali/server3.py r,
  /usr/bin/python3.9 ix,
}
```

Listing 3. Profile created by *AppArmor* for a simple Python web server

mand is suitable for small applications. Systemic profiling is suitable for creating profiles for many programs that make up an application or for applications that continue running on the computer for a long time. Systemic profiling requires the user to follow the steps shown in Figure 7 [26].

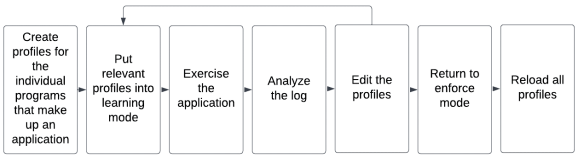

Figure 7. Systemic profiling method in *AppArmor* [26]

Systemic profiling method requires more efforts from users to define a complete profile for an application. However, the standalone profiling method is not perfect as it suffers from some important issues. To illustrate these issues, we decided to define a profile for a simple Python script that allows us to run a web server. This server requires *cap_net_bind_service* capability. Listing 3 below shows an example of an *AppArmor* profile that was created for the simple Python web server using the learning mode. The user has to add to the profile not only the privileges to assign to the Python script, but also a set of traditional permissions needed by the Python web server.

The first issue that we have identified is that the user has to run the Python script using *sudo* in order to define completely and correctly the needed profile. Running a tool with full administrative privileges is not recommended in case the tool has known vulnerabilities or is not trusted. When comparing *AppArmor* with *RootAsRole*, the *capable* command that we have designed does not require the user to run a program with the *sudo* command to know the needed capabilities. The *capable* command can know the capabilities by intercepting *cap_capable()* syscall in the Linux kernel and reporting back the information to the user space to indicate the capability required by the tested program.

The second issue that we have noticed is related to the difficulty to know the processes that are privileged for a system that is using *AppArmor*. On one hand, the profiles created by *AppArmor* using the learning mode are not

```
# vim:syntax=apparmor

# This file contains basic permissions for Apache
    and every vHost

  abi <abi/3.0>,

  include <abstractions/nameservice>
```

Listing 4. AppArmor profile for <abstractions/apache2-common>

always mentioning explicitly the capabilities used by one program. For example, the Python script for the simple web server requires the use of the *cap_net_bind_service* capability, but the generated profile does not explicitly show that the capability has been added. On the other hand, the user cannot also trust the information given by the Permitted & Effective sets of the process that is running the Python script, as it shows that it has the full list of Linux capabilities, when in fact the script has only the *cap_net_bind_service* capability.

More seriously, *AppArmor* does not allow to apply the least privilege principle strictly as it adds many rules that are not really needed by the Python script. Listing 3 shows that the profile contains a set of *include* statements (such as "*include* <abstractions/apache2-common>") that are used to pull in components of other profiles created by *AppArmor* and by that it also retrieves access permissions of other programs. Users will not be able to verify the privileges by checking the profile files as the capabilities that are included in the abstraction files are not usually verified by users. Those *include* statements make it hard for the user to know what permissions are being added and whether they are needed or not. In the profile shown in Listing 3, "*include* <abstractions/ apache2-common>" is explicitly listed in the profile. However, the user can't know what privileges are retrieved unless they refer to the *AppArmor* GitLab page [27]. As shown in Listing 4 the user will find out that another include statement is provided to include yet another set of permissions which is "*include* <abstractions/nameservice>". Upon examining the page detailing nameservice profile, the user can finally note that more *include* statements are added as shown in Listing 5 which enters a user in a loop trying to identify what privileges and permissions are being given to the program by *AppArmor*.

grsecurity adopts like *AppArmor* a pathname enforcement approach. *grsecurity* implements the Role Based Access Control (RBAC) model [20] that allows an administrator to determine for each program the authorized users and their associated roles as well as the access permissions on the resources that programs can access. It also determines the list of Linux capabilities that any programs have the authorization to take. However, *grsecurity* is not only an LSM, it is presented as an extensive security enhancement to the Linux kernel because it proposes a set of protection measures against a set of well-known security threats.

SELinux implements a label-based access control (LBAC) model [23] and uses the extended attributes to store

```
# nis
include <abstractions/nis>

# ldap
include <abstractions/ldapclient>

# winbind
include <abstractions/winbind>

# likewise
include <abstractions/likewise>

# mdnsd
include <abstractions/mdns>

# kerberos
include <abstractions/kerberosclient>

#libnss-systemd
include <abstractions/nss-systemd>
```

Listing 5. AppArmor profile for nameservice

label values. As a consequence, *SELinux* suffers from the same problems that we mentioned earlier with regard to the extended attributes. In addition, *SELinux* is very hard to manage even by experienced administrators due to the difficulties with regard to the management of the LBAC system.

Theoretically, Linux administrators can use these modules to prevent the privilege escalation problem. However, only a subset of the processes is confined, while the other applications are left unconstrained. In addition, many Linux administrators may not need to extend their kernels with complex LSM but they may wish to restrict the distribution of Linux privileges without having to define a complete policy that includes the traditional permissions. Our module *RootAsRole* allows Linux users to restrict the administrative privileges and can be used simultaneously with LSM.

4. RootAsRole Module

The *pam_cap*[8] module is the only module that can be used by administrators to restrict capabilities in a similar way to *RootAsRole*. However, this module has three major problems: the use of extended attributes to store the capabilities, no fine-grained distribution of capabilities to Linux users, and no tool that can help Linux users to figure out the capabilities requested by a program.

To solve these problems, we have created the *RootAsRole* module. *RootAsRole* comes with a set of commands to handle the administrative privileges: *addrole*, *deleterole*, and *editrole* commands are used by Linux administrators to add, delete and edit roles in a central configuration file, where each role is a set of administrative privileges. *RootAsRole* gives the possibility to Linux administrators and users to discover the capabilities that are requested by a binary file when being executed through the *capable* command. Finally, *RootAsRole* allows Linux users to assign the defined roles to themselves through the *sr (switch role)* command. This command provides a set of options that allow users to know what roles are currently assigned to themselves in the configuration file, and what capabilities are assigned to each role.

An administrator defines the rules that allow the fine-grained distribution of capabilities to users and groups, without needing to inject the capabilities into the extended attributes of binary files. Each time a Linux user wants to execute a program that necessitates one or more capabilities, they should assign the role to themselves that contains the needed capabilities, providing there is a rule that allows it.

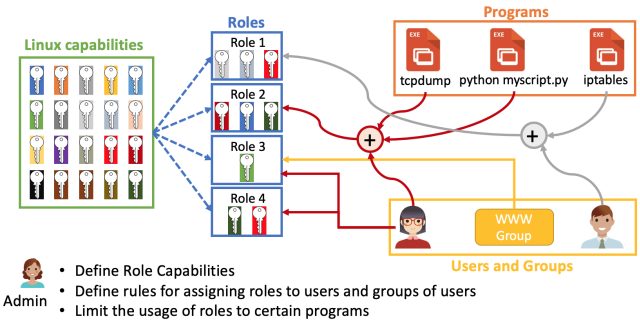

Figure 8. Role definition in *RootAsRole* module.

The *RootAsRole* module comes with a configuration file called capabilityrole.xml that is stored in /etc/security. This file allows an administrator to define each role along with its list of capabilities, as well as the lists of users, groups, and programs that can be assigned this role. In addition, users and groups can be constrained as to when the role can be assigned. For example, Listing 6 shows a configuration file that defines the role *role1*. This role has two capabilities (*cap_net_raw* and *cap_net_admin*). Users *tomas* and *remi* can be assigned this role as can be the users of groups *adm* and *office*. However, *remi* can only be assigned this role when he is executing the *tcpdump* command. It should be noted that the administrators must indicate the full path of the constrained binary (i.e. *tcpdump*) in the configuration file. During the execution, users have to indicate also the full path of the constrained binaries. The other users (user *tomas* and members of groups *adm* and *office*) are not constrained and can be assigned this role with any program; they get a privileged shell inside which they can run any program (that only needs the user's assigned capabilities).

The configuration file of *RootAsRole* limits the use of *xattrs*. It is read into a central database that allows administrators to keep track of the assigned roles and the programs that users can use with these roles.

When the administrator lists a program under a user, the user can only run this program with the capabilities. In some cases, a conflict may exist between the programs defined at the user level and those defined at the group level. For example, administrators may restrict a user to execute only one program, but at the same time, they authorise the user's group to execute any program. In these situations, we

8. https://sites.google.com/site/fullycapable/pam_cap-so

```
<configuration xmlns="http://mycapconf.xml">
 <roles>
  <role name="role1">
      <capabilities>
        <capability>cap_net_raw</capability>
        <capability>cap_net_admin</capability>
      </capabilities>
      <users>
        <user name="tomas"/>
        <user name="remi">
         <commands>
           <command>/bin/sbin/tcpdump</command>
           <args>
              <arg>-i eth0</arg>
           </args>
         </commands>
        </user>
      </users>
      <groups>
        <group name="adm"/>
        <group name="office"/>
      </groups>
   </role>
</roles>
</configuration>
```

Listing 6. Example definition of a role in the configuration file.

consider the configuration at the user level overrides the configuration at the group level i.e. the specific rule overrides the generic rule. In the example above, the configuration shows that *remi* can only run the *tcpdump* program, but if *remi* was a member of the *adm* group, which can run any program, this would not apply to *remi*.

4.1. Motivation scenarios

We will demonstrate the use of our *RootAsRole* module through two different scenarios. These scenarios show the advantage of our module with regards to the *pam_cap* module which is used today to manage Linux capabilities.

- Running privileged Python scripts (Scenario 1): A user contacts an administrator to ask for a privilege that allows running a HTTP server that the user developed using Python. The script needs the privilege *cap_net_bind_service* to bind the server socket to port 80. Without our module, the administrator has two options: (1) Use the *setcap* command to assign the capability to the Permitted set of the Python interpreter or (2) use the *pam_cap* module to assign the *cap_net_bind_service* to the user and then inject this capability into the Inheritable and Effective sets of the Python interpreter. Both solutions pose security risks. In the case of option (1) the Python interpreter can be used by any other user who will automatically gain the *cap_net_bind_service* privilege. In the case of option (2), all other Python scripts run by the user will have the same privilege. It is not possible to run other Python scripts without giving them the privilege *cap_net_bind_service*.
- Preloading shared libraries (Scenario 2): Suppose developers want to test a library that they have developed in order to reduce the downloading time on their server. The developers should use the *LD_PRELOAD* environment variable to load the shared library that intercepts all network calls made by the servers' processes. With the current capabilities' tools, the administrator can use the *setcap* command or *pam_cap.so* to give the developers' library and test program the *cap_net_raw* capability. However, the developers still cannot run their test program because, for security reasons, the Linux Kernel does not load the libraries defined in *LD_PRELOAD* for a program that has capabilities in its extended attributes.

4.2. Scenario 1: Running privileged Python scripts

To demonstrate the implementation of Scenario 1, we selected a Python script, server.py, which can be used to run a HTTP server [5]. When executing the server.py script without any privileges, we get the '*permission denied*' error message (Figure 9).

```
tomas@osboxes:~$ python3 server.py -p 80
Traceback (most recent call last):
  File "server.py", line 8, in <module>
    with socketserver.TCPServer(("", PORT), Handler) as httpd:
  File "/usr/lib/python3.8/socketserver.py", line 452, in __init__
    self.server_bind()
  File "/usr/lib/python3.8/socketserver.py", line 466, in server_bind
    self.socket.bind(self.server_address)
PermissionError: [Errno 13] Permission denied
```

Figure 9. Run the Python HTTP server by a normal user.

In order to determine what privileges are needed by the script server.py, we use the tool *capable* as in Figure 10

```
tomas@osboxes:~$ capable -c "python3 server.py -p 80"
Traceback (most recent call last):
  File "server.py", line 8, in <module>
    with socketserver.TCPServer(("", PORT), Handler) as httpd:
  File "/usr/lib/python3.8/socketserver.py", line 452, in __init__
    self.server_bind()
  File "/usr/lib/python3.8/socketserver.py", line 466, in server_bind
    self.socket.bind(self.server_address)
PermissionError: [Errno 13] Permission denied

Here are all the capabilities intercepted for this program :
cap_net_bind_service, cap_sys_admin
WARNING: These capabilities aren't mandatory, but they can change the behavior of tested program.
WARNING: CAP_SYS_ADMIN is rarely needed and can be very dangerous to grant
```

Figure 10. Using the *capable* tool to determine the required privileges.

This tool tells us that the script requires the capabilities: *cap_net_bind_service* and *cap_sys_admin*. The capability *cap_sys_admin* will be always returned because the shell process is using *fork()* syscall to run any process. The capability *cap_sys_admin* is required by *fork()* syscall but it can be sometimes required by the program itself, that's why we recommend always testing the program without *cap_sys_admin*. The script is working when tested with the capability *cap_net_bind_service*, so we conclude that this script requires only this capability to bind the server to port 80.

When an administrator runs the script using the *sudo* command (*sudo python server.py -p 80*), the Python process is given the full set of privileges, because as indicated earlier, any process whose *euid* or *egid* is equal to zero automatically gets the full set of privileges (i.e. emulating

mode). Figure 11 shows that process 22325 has the full set of Permitted, Effective and Bounding capabilities as these values correspond to the first 38 bits being set to 1.

```
tomas@osboxes:~$ cat /proc/22325/status
Name:   python3
Threads:        1
SigQ:   0/7740
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000001001000
SigCgt: 0000000180000002
CapInh: 0000000000000000 Inheritable
CapPrm: 0000003fffffffff Permitted
CapEff: 0000003fffffffff Effective
CapBnd: 0000003fffffffff Bounding
CapAmb: 0000000000000000 Ambient
```

Figure 11. Values of the capability sets for the Python process launched by the *sudo* command.

When administrators use the *setcap* command to assign the *cap_net_bind_service* capability to the Python interpreter (Figure 12) they create a security risk because now all other users of the system will be able to run Python scripts with the same privilege.

```
tomas@osboxes:~$ sudo setcap cap_net_bind_service+ep /usr/bin/python3.8
tomas@osboxes:~$ python3 server.py -p 80
serving at port 80
```

Figure 12. Setting the Permitted and Effective capabilities for the Python3.8 interpreter.

The administrators can alternatively use the *pam_cap* module by first setting the *cap_net_bind_service* in the /etc/security/capability.conf file (*pam_cap*'s configuration file), and then using *setcap* command as in Figure 13.

```
tomas@osboxes:~$ sudo setcap cap_net_bind_service+ie /usr/bin/python3.8
tomas@osboxes:~$ python3 ./server.py -p 80
serving at port 80
```

Figure 13. Setting the Inheritable and Effective capabilities for the Python3.8 interpreter.

This solution raises another potential security risk because now any Python script run by the same user will obtain the same privilege.

Our solution avoids these security risks. Suppose that the capabilityRole.xml contains the configuration shown in Listing 7

In this case, the user *tomas* can use the role *role2* by using the *sr* command whenever the HTTP server Python script (server.py) is run – see Figure 14.

As shown in Figure 15, the user *tomas* cannot run another script with the same privilege. Generally, administrators may not want to limit the use of capabilities to certain scripts or programs, as this would create a lot of work for them. However, in some cases administrators may need this option especially when they do not completely trust all their users.

```
<role name="role2">
      <capabilities>
        <capability>
           cap_net_bind_service
        </capability>
      </capabilities>
      <users>
        <user name="root"/>
        <user name="tomas">
          <commands>
                <command>
                   python3 /home/tomas/server.py
                </command>
          </commands>
        </user>
      </users>
      <groups>
        <group name="adm"/>
        <group name="office"/>
      </groups>
</role>
```

Listing 7. Defining *role2* for running the Python HTTP server.

```
tomas@osboxes:~$ sr -r role2 -c "python3 /home/tomas/server.py"
Authentication of tomas...
Password:
Privileged bash launched with the role role2 and the following capabilities : cap_net_bind_service.
serving at port 80
```

Figure 14. User *tomas* assigns the role *role2* to execute the Python HTTP server.

```
tomas@osboxes:~$ sr -r role2 -c "python3 /home/tomas/anotherprivilegedscript.py -p 80"
Authentication of tomas...
Password:
This role and command cannot be used with your user or your groups: Permission denied
tomas@osboxes:~$
```

Figure 15. User *tomas* unable to run another Python script.

4.3. Scenario 2: Interception of socket() calls

To demonstrate the implementation of Scenario 2, we selected the previously published example code [6] that intercepts any *socket()* call. Listing 8 shows a simple client code that opens a raw socket. Listing 9 shows the library code that tries to intercept the *socket()* call. When it is successful, it prints out the message *socket() call intercepted*.

When executing the *simple client* code without any privileges, we get the expected message *"permission denied"*. We can use the *capable* command to determine the privileges that are requested by the *simple client* program (see Figure 16).

```
tomas@osboxes:~$ capable -c ./simpleclient
Error: couldnot create socket
: Operation not permitted

Here are all the capabilities intercepted for this program :
cap_net_raw, cap_sys_admin
WARNING: These capabilities aren't mandatory, but they can change the behavior of tested program.
WARNING: CAP_SYS_ADMIN is rarely needed and can be very dangerous to grant
```

Figure 16. Results of using the *capable* tool.

As can be seen, this program requests the *cap_net_raw* and *cap_sys_admin* capabilities. Since the *capable* tool

```c
/*
* A simple client that open a socket
*/

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/tcp.h> //Provides declarations
    for tcp header
#include <netinet/ip.h> //Provides declarations
    for ip header

int main(int argc, char *argv[]){
  int sockfd;

  //Create socket and check for error
  if((sockfd = socket(AF_INET, SOCK_RAW,
    IPPROTO_TCP)) < 0)
  {
     perror("Error : could not create socket\n");
     return 1;
  }else{
     printf("Socket successfully created\n");
  }
  return 0;
}
```

Listing 8. The simple client C code to create a socket.

```c
/*
* A simple socket hook which calls the original
    socket library
*/

#define _GNU_SOURCE
#include <stdio.h>
#include <sys/socket.h>
#include <dlfcn.h>

int (*o_socket)(int,int,int);

int socket(int domain, int type, int protocol){
  //find the next occurence of the socket()
    function
  o_socket = slsym(RTLD_NEXT, "socket");

  if(o_socket == NULL)
  {
     printf("Could not find the next socket()
    function occurence");
     return -1;
  }

  printf("socket() call intercepted\n");

  //return the result of the call to the original
    C socket() function
  return o_socket(domain,type,protocol);
}
```

Listing 9. The socket_hook interception code.

recommends running the program without *cap_sys_admin*, then we would run the program with only the *cap_net_raw* capability assigned, and it would successfully open the socket. However, when the *LD_PRELOAD* is configured, the Linux kernel disables the interception of the *socket()* call because the program to be intercepted has capabilities in its extended attributes. Figure 17 shows how the interception works correctly when the capability is removed from the *simple client* file (using *setcap -r*) – the operation is denied, and not correct when the capability is added to it (it creates a socket without being intercepted). As a consequence, the only way to realise the interception and to open up an intercepted socket is to use the *sudo* command, which we do not want to do.

```
tomas@osboxes:~$ sudo setcap -r ./simpleclient
tomas@osboxes:~$ LD_PRELOAD=./socket_hook.so ./simpleclient
socket() call intercepted
Error: couldnot create socket
: Operation not permitted
tomas@osboxes:~$ sudo setcap cap_net_raw+ep simpleclient
tomas@osboxes:~$ LD_PRELOAD=./socket_hook.so ./simpleclient
Socket success fully created
tomas@osboxes:~$
```

Figure 17. Interception is stopped when capabilities are injected into the extended attributes of simple client.

By using our *RootAsRole* module, we can realize the interception and open the socket by giving only the *cap_net_raw* capability to the *simple client* program. Figure 18 shows how the interception program works correctly when the user *tomas* uses the role *role1* (defined in Listing 6).

```
tomas@osboxes:~$ sr -r role1
Authentication of tomas...
Password:
Privileged bash launched with the role role1 and the following capabilities : cap_net_raw.
~:tomas (role1)$ LD_PRELOAD=./socket_hook.so ./simpleclient
Socket success fully created
~:tomas (role1)$
```

Figure 18. Interception works correctly with our *RootAsRole* module.

5. The implementation

In the section, we are describing how *sr*, *capable* and role manager tools have been internally implemented. *sr* was built based on the existing logic of Linux capabilities, while *capable* was built based on eBPF technology that allows to intercept the syscalls and report their arguments to the user space. Finally, the role manager tools are defined to help users add/edit/remove roles without having to edit manually the XML configuration file of *RootAsRole*.

5.1. Implementing the *sr* command

The implementation of the *sr* command depends on the Ambient capability set that has been added to the Linux kernel since Linux 4.3 [12]. The objective of the *sr* command is to allow users to assign roles to themselves and to list all the

roles available to them, along with the capabilities assigned to each role. The command has the following parameters:

- -r, --role=<*the role to use*>
- -c, --command=<*the program to launch with the role*>
- -n, --no-root = <*execute the bash or the command without any special treatment of the root user (uid 0). This option allows the user to deactivate the emulation mode of the Linux kernel*>
- -u, --user=<*substitute user reserved for administrators for service management*>
- -i, --info =<*print the roles and their capabilities available to the user, or when used with the –r parameter the commands the user is able to process with this role*>
- -v, --version = <*print the version of RootAsRole*>
- -h, --help =<*print this help*>

Implementing this command must not necessitate any modifications to the kernel. Furthermore, the executable file of *sr* must have as few capabilities as possible, so that only the capabilities from the user's role are used. In addition, it must offer the same level of usability as the *sudo/su* commands so that Linux users will find it easy to adopt. As a consequence, our implementation must fulfil the following requirements:

Requirement 1: calculate the capability sets after an exec call

The Ambient set was added to the Kernel in Linux 4.3 to resolve problems with using the extended attributes for storing file capabilities. The following rules are now applied by the kernel to calculate the capabilities of a new process P' that is created when a process P makes an exec call to an executable file F:

$$\begin{aligned} P'_{Ambient} &= (file\ \ is\ \ privileged)?0 : P_{Ambient} \\ P'_{Permitted} &= (P_{Inheritable} \& F_{Inheritable}) \\ &\quad |(F_{Permitted} \& P_{Bounding}) \\ &\quad |P'_{Ambient} \\ P'_{Effective} &= F_{Effective}?P'_{Permitted} : 0|P'_{Ambient} \\ P'_{Inheritable} &= P_{Inheritable} \\ P'_{Bounding} &= P_{Bounding} \end{aligned} \tag{2}$$

Where as before: & is bitwise AND, | is logical OR, z=x? y:w means if x has a value or true; z will take the value of y otherwise z will take the value w.

As the rules show, a new process has three options for getting its Permitted capabilities:

- by masking the Inheritable set of the calling process with the file's inheritable capability set;
- by masking the Permitted set of the file with the Bounding set of the calling process; or
- from its own Ambient set (which is copied from the Ambient set of the calling process if the file is not privileged).

Clearly, the third option has the advantage if the executable file does not have any extended attributes. The resulting process P' copies the Ambient set of P into its Ambient set, that in turn is copied into its Permitted and Effective sets. The Inheritable and Bounding sets of the resulting process P' are copied from the respective sets of the calling process P. In the case where the binary file is a privileged file, the rules are the same as before. This is to ensure backward compatibility with the old capability model before the addition of the Ambient set.

It should be noted these rules are not applied when a process creates a child process via the *fork()* call as the child process inherits exact copies of its parent's capability sets.

Requirement 2: Constraining the Permitted set

A process cannot add any new capability to its own Permitted set. However, it can remove some capabilities from its own Permitted set.

Requirement 3: Constraining the Effective set

A process can add and remove capabilities to its Effective set, as long as these capabilities are present in its own Permitted set. Good programming practice is to activate capabilities when needed by adding them into the Effective set, and de-activating them when not needed by removing them from the Effective set.

Requirement 4: Constraining the Inheritable set

The Inheritable set of a process should normally be a subset of its Permitted set. However, a process can add additional capabilities in its Inheritable set if it has the capability *cap_setpcap* in its Permitted set.

Requirement 5: Constraining the Ambient set

A Process can add a capability to its Ambient set only when the capability is present simultaneously in its Permitted and Inheritable sets. Thus, the intersection of the Permitted and Inheritable sets is considered to be the superset of the Ambient set. A new option, called *PR_CAP_AMBIENT* has been added to the *prctl()* syscall in order to use this feature.

Requirement 6: Filling the capability sets of an executable file

A process can add capabilities to the capability set of an executable file only when it has the capability *cap_setfcap* in its Effective set.

Before using the *sr* command, the system administrator must edit the capabilityrole.xml configuration file to indicate the roles that can be assigned by users or groups of users, and with which programs those users or groups of users can use those roles.

When calling the *sr* command, the user must indicate the role to be assigned. The users can invoke the --info option to learn which roles they can assign to themselves. In addition, the users can indicate which program they want to use with the requested role. In the current implementation, the users cannot assign more than one role, although this could be a useful future enhancement. If the users do not indicate a program, the *sr* command will launch a privileged shell that contains the capabilities of the requested role.

How can the *sr* command launch the user's program and inject the capabilities of the user's role into the new process if the *sr* command does not possess any privileges? One

option would be to ask the user to run the *sr* command using the *sudo* command. In this case, the *sr* process will have the full set of Linux capabilities and can inject the capabilities of the user's role into the extended attributes of the user's program and then run it. But this defeats the objective of our research, which is to provide an alternative to the *sudo/su* commands that give programs more privileges than they need. In addition, we do not want to modify the extended attributes of the user's program because another user may use this program at the same time. Instead, we give the *sr* program file the minimum necessary capabilities at installation time, as described below. On invocation, the *sr* process then creates a temporary binary file called *sr_aux* and injects the user's requested capabilities into the Inheritable and Permitted sets of this. After that, the *sr* process executes *sr_aux* to produce an auxiliary process that has the *user's* capabilities in its Permitted and Inheritable sets. Then, according to requirement 5 it will be able to add these capabilities to its Ambient set, and to execute either the program that has been selected by the user through the optional -c parameter, or start a shell.

Figures 24 and 25 in the Appendix provide detailed flow charts for the implementation of *sr* and *sr_aux* respectively. The *sr* process checks whether the role exists and whether the administrator has authorised the user or its group to assign the requested role with the requested program. If authorized, then *sr* authenticates the users by asking them to provide their password. When the authentication is successful, *sr* will prepare the program or shell to run with the requested capabilities.

According to Requirement 2, the *sr* process cannot add the list of capabilities requested by the user to its own Permitted set and cannot add them to its Ambient set either (because they are not present in its Permitted set). It, therefore, creates the *sr_aux* temporary file to achieve this. In order to fill the Permitted set of *sr_aux*, we have to use the extended attributes of *sr_aux* and inject the role's capabilities into these attributes. According to Requirement 6, *sr* needs the capability *cap_setfcap* in its Permitted set in order to be able to inject capabilities into the extended attributes of the *sr_aux* file. Consequently, we add the *cap_setfcap* capability into the Permitted set of the *sr* file at installation time. The *sr* file is now privileged. According to Requirement 1, filling the Inheritable set of the *sr_aux* file will not copy them into the Inheritable set of the resultant process. Instead, they have to be in the Inheritable set of the *sr* process. According to Requirement 4, *sr* needs the capability *cap_setpcap* in order to add capabilities into its Inheritable set when they are not present in its Permitted set. Thus, we also have to add the *cap_setpcap* capability to the Permitted set of the *sr* file during installation.

Now the user's requested capabilities can be added to the Permitted set of the *sr_aux* file and to the Inheritable set of the *sr* process. According to Requirement 1, when *sr* makes an exec call to *sr_aux* the capabilities of the file's Permitted set will be copied into the Permitted set of the resultant process and the capabilities of the Inheritable set of the *sr* process will be copied into the Inheritable set of the resultant process of *sr_aux*. According to Requirement 5, the resultant process of *sr_aux* can dynamically add the user's list of requested capabilities into its own Ambient set. Finally, the resultant *sr_aux* process makes an exec call to the program requested by the user (or the shell). Then, according to the rules of Requirement 1, the user's resultant process will have the requested capabilities in its Permitted and Effective sets. As a consequence, the Kernel will authorise the resultant process to perform the privileged operation requested by the process.

Nevertheless, one major security flaw still persists. To be able to assign proper Inheritable capabilities, *sr* should have the full bounding capabilities. As $P'_{Bounding} = P_{Bounding}$, *sr_aux* will also get the full bounding capabilities. Consequently, the executed child command will get the full bounding capabilities. With that, a malicious user can create a stub program and utilizes the proposed *RootAsRole* model to run it with full bounding capabilities. The resulting stub process is now capable of executing any child command and granting it full capabilities via its Inheritable set bypassing the whole security policy. To address this issue, *sr_aux* has to drop its own bounding capabilities or at least mask it with its own permitted ones, before executing the requested command. The source code of our module is published under the GPL License, and can be found here [7].

5.2. Implementing the *capable* command

The implementation of the *capable* command was not straightforward because the kernel does not offer any service that can tell the user what kind of capabilities a program needs. Consequently, we had to use the Linux Extended BPF APIs [13] to build the *capable* command and provide it with a filtering mechanism.

User space programs use a list of available syscalls (e.g. *fork*, *openat*, *read*, *write*, *setuid*, etc.). All these syscalls include several checks in their code to verify whether the calling process has the required privilege or not. For example, the *setuid()* syscall checks whether the calling process has the capability *cap_setuid*. If the calling process does not have this capability in its Effective set, the *setuid()* code returns the error code *"EPERM"* (see errno(3) [4]), which is always interpreted by the User space process as permission denied. The kernel does not tell the calling process that *cap_setuid* is needed. This issue represents a major obstacle to the adoption of Linux capabilities, because administrators and users do not have commands that can tell them what kind of capabilities the used programs are asking for. We have built the *capable* command to address this issue.

There are two functions that are called by Linux syscalls in order to check whether a process has the required capability or not (Figure 19). The first function is called *capable()* and the second function is called *ns_capable()*. The difference between them is that *ns_capable()* has the ability to check the capability in a namespace. However, both of them are implemented by a function called *cap_capable()* in the capability LSM.

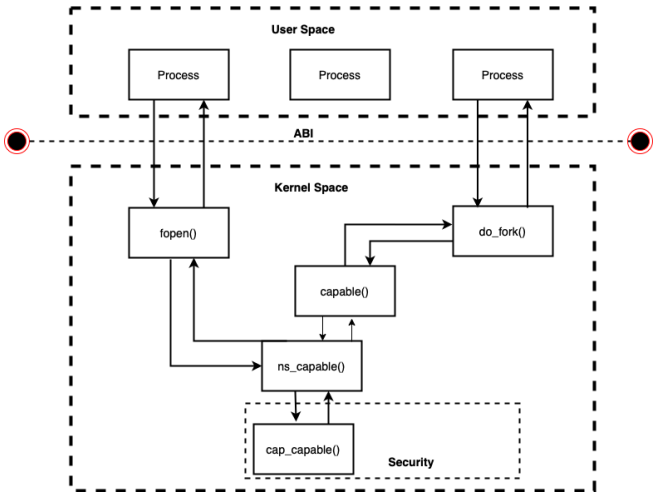

Figure 19. Relation between *capable*, *ns_capable* and *cap_capable*.

As Listing 10 shows, *cap_capable()* checks whether the calling process has the required capability in its Effective set. If not, it returns the *"EPERM"* error code without indicating the required capability to the calling process. One way to resolve this issue would be to modify the *cap_capable()* source code. However, discovering the impact of this modification is a very difficult task. Consequently, we preferred to opt for another solution, specifically, using one of the Linux tracing technologies such as Kprobe [14], tracepoints [15] or Ftrace [16] with eBPF [13].

The basic idea of all the tracing technologies is to insert trace points in specific locations of the kernel. When the processor executes one of these points, a trace event is generated. This event contains the execution context information that is stored in the registers of the processor. We used Kprobe [14] to get the context execution whenever *cap_capable()* is executed by the kernel. This allowed us to get the arguments of the *cap_capable()* function, which are:

- *cred*: this provides the id of the calling process, and
- *cap*: this provides the capability that is requested by the calling process.

However, using Kprobe is not easy for an average user. Indeed, the user has to read a huge amount of events because Kprobe returns the *cap_capable()* events for every process in the system. To resolve this issue, we use eBPF [13]. This allows us to obtain only the capability required by one program without needing to read the whole list of events.

eBPF is an extension to the Berkeley Packet Filter (BPF) that was designed to filter network packets by handling them in a virtual machine inside the kernel. Alexei Starovoitov from Facebook extended the original BPF to include 64-bit registers and increase the set of instructions [17]. The basic idea consists of building two programs: the Kernel program that is attached to the execution path in the Linux kernel e.g. by linking it to Kprobe; and the User program that loads the Kernel and reads the data returned by it. The eBPF virtual machine includes a verifier that checks

```
int cap_capable(const struct cred *cred, struct
    user_namespace *targ_ns,
    int cap, unsigned int opts)
{
  struct user_namespace *ns = targ_ns;

  /* See if cred has the capability in the target
    user namespace by examining the target user
    namespace and all of the target user namespace
    's parents.
  */
  for (;;) {
    /* Do we have the necessary capabilities? */
    if (ns == cred->user_ns)
      return cap_raised(cred->cap_effective, cap)
    ? 0 : -EPERM;

    /*
     If we're already at a lower level than we're
    looking for, we're done searching.
    */
    if (ns->level <= cred->user_ns->level)
      return -EPERM;

    /*
     The owner of the user namespace in the parent
     of the user namespace has all caps.
    */
    if ((ns->parent == cred->user_ns) && uid_eq(ns
    ->owner, cred->euid))
      return 0;

    /*
     If you have a capability in a parent user ns,
     then you have it over all children user
    namespaces as well.
    */
    ns = ns->parent;
  }

  /* We never get here */
}
```

Listing 10. Code source of *cap_capable()* [30].

the loaded Kernel part before accepting it, to ensure that it does not contain an endless loop or does not try to access unauthorised data in memory. eBPF also provides a set of function helpers to access memory in a safe way. For example, *bpf_probe_read* is a safe version of the memcpy function that copies characters from one area to another. In addition, eBPF provides data structures, called maps, that are accessible from the user space of the eBPF program. Maps are important because the eBPF verifier does not allow more than 512 bytes to be allocated in the stack.

Maps are created in the Kernel and filled with Kprobe events whenever a call is made to *cap_capable()*. We provided a filtering mechanism to show to users only the list of capabilities that are requested by their program. It is based on the idea of running the user's programs in a dedicated PID namespace [18]. This provides isolation protection to the user's system when the tested program is unknown, and it allows the list of processes that have been run in the dedicated namespace to be identified. The User

program shows the user the list of capabilities that have been requested by the list of processes executed in the dedicated PID namespace.

Our *capable* tool always returns the capability *cap_sys_admin*, even though the tested program does not explicitly require this capability. This comes from the fact that the *fork()* syscall checks the presence of the *cap_sys_admin* capability whenever it is called. Whenever a program is executed by the user, the *fork()* syscall is called by the kernel in the context of the new process that represents the tested program. As a consequence, our tool always returns *cap_sys_admin* because we cannot see the difference between when this capability is called explicitly by the program or through the *fork()* call.

5.3. Implementing the Role Manager tool

The fact the administrators have to edit manually the central configuration file can create different issues, especially when the number of roles becomes very big. To handle this issue, we have implemented a set of tools that allow the system administrators to easily edit the central configuration file (capabilityRole.xml). The tools allow to maintain the integrity of the XML file, reduce errors and automate tasks such as adding roles, editing roles or deleting roles. The *addrole* command will enable the system administrator to add an additional role to the configuration file along with a list of privileges, users, groups, authorized commands, and authorized programs. The following example illustrates the use of the *addrole* command to add the role *role3*. In a system that includes a user *tomas* and a group titled *zayed*, the system administrator might require the *cap_fowner* and the *cap_setuid* capabilities to be assigned to the specified user and specified group. This role cannot be used with any program, so the administrator restricts this role to execute only the commands: */usr/bin/passwd*, */usr/bin/chmod* and */opt/myprogram -i*. The system administrator can simply use the *addrole* command as shown in Figure 20. This command

```
tomas@osboxes:~$ sudo addrole role3 cap_fowner,cap_setuid -u tomas -g zayed -c "/usr/bin/passwd" -c "/usr/bin/chmod" -c "/opt/myprogram -i"
```

Figure 20. *addrole* command

will add the role to the configuration file to the assigned user, authorized group and the authorized commands as shown in Listing 11.

The tool also includes an *editrole* command that would redirect the administrator to choose whether they would like to add, edit or delete a specific information of an existing role. For example, the system administrator can add a capability to an existing role. Figure 21 shows the command required to add the *cap_net_raw* capability to *role3*.

The *editrole* command can also be used to edit an existing node to edit the username, group name, etc. It can also be used to delete specific information from an existing role, such as removing a user. Finally, the tool includes a *delete* role command that allows the user to delete a

```
<role name="role3">
      <capabilities>
        <capability> cap_fowner </capability>
        <capability> cap_setuid </capability>
      </capabilities>
      <users>
        <user name="tomas"/>
      </users>
      <groups>
        <group name="zayed"/>
         <commands>
          <command>/usr/bin/passwd</command>
          <command>/usr/bin/chmod</command>
          <command>/opt/myprogram -i</command>
         </commands>
      </groups>
</role>
```

Listing 11. *role3* added to the configuration file.

```
tomas@osboxes:~$ sudo editrole role3
1. Add
2. Edit
3. Delete
0. Quit
What do you want to do ? -> 1
Use URL syntax for add an element to xml file
Example : /capabilities/cap_net_bind_service
What do you want to add ? -> /capabilities/cap_net_raw
```

Figure 21. *editrole* command

previously added role. In addition to the above-mentioned commands, The tools provide additional verification to avoid any possible errors, for example, the user would get an error if they attempt to add a role that already exists in the configuration file. The Role management tools also verify the user input by controlling which nodes the user is allowed to edit as shown in Figure 22. The user is not allowed to edit nodes that would affect the validity of the XML configuration file.

```
tomas@osboxes:~$ sudo editrole role3
1. Add
2. Edit
3. Delete
0. Quit
What do you want to do ? -> 2
1 role3 :
2       Capabilities :
3               cap_fowner
4               cap_setuid
5       Users :
6               tomas
7       Groups :
8               zayed
9                       Commands :
10                              /usr/bin/passwd
11                              /usr/bin/chmod
12                              /opt/myprogram -i

Use the displayed tree and selects the number corresponding to the node -> 5
Requested node invalid, retry ->
```

Figure 22. *editrole* command providing user input verification

6. Complexity and Empirical Performance Analysis

RootAsRole provides thus a balance between the fine grained - but complex to handle - LSM offer, and the often

too coarse grained policy given by *sudo*. One important aspect of these systems is their high frequency of use, through manual terminal or through batch processing. Thus, one key factor of comparison is the scalability of the tool: its capacity to increase its processing time smoothly as the configuration grows. We provide here a theoretical and empirical comparative analysis of our proposition against *sudo*, since both tools share the same structure of process.

Indeed, *sudo* and *sr* (the main application used in *RootAsRole*) present the same conceptual steps: they (i) authenticate the user if required, (ii) read the configuration file to verify whether the command to execute with the requested user/role is authorized (and/or to extract the role name to use for *sr*), and eventually (iii) execute the requested command. As the definition of the policy increases, one bottleneck for both programs can appear in the second step that aims, regarding the parsing of the configuration file.

We reviewed the source code of *sudo* (the latest stable 1.9.1 version) and *sr* to compare their computational complexity. Both programs appear to have an equivalent theoretical efficiency, with an order of $O(n)$ complexity, with n the number of items in the configuration file. For *sudo*, an item is either a user or a command while for *sr* it is either a role, a user or a command. While this analysis advocates for an equivalent performance for both *sudo* and *sr* programs as they both expose a linear efficiency, it does not give any clue on the differences between their linear execution time: (i) the constant time required for any configuration file, and (ii) the extra amount of time required with the increasing of the configuration. Since important differences exist between *sudo* and *sr* that can have an impact on performance, we decided to carry on with an empirical comparative analysis.

We followed the given protocol that aims at measuring execution time in real condition:

- for a given number U of users/roles and C commands, generate of configuration file of $U \cdot C$ items for both *sudo* and *sr*. The last command of the last user/role in the configuration file matches the test command (i.e.: *whoami*) while the last user/role is the test one;
- using the test user, execute the test command with *sudo* and measure its execution time with the GNU time command;
- invalidate user's cached credentials (*sudo -k*) for the next test;
- execute the test command with the test role through *sr* and measure its execution time with the GNU time command.

This protocol was implemented in a shell script to automate the procedure, including the authentication processes. For both commands we measured 3 different times: the real execution time, the CPU-seconds spent in kernel mode, and the CPU-seconds spent in user mode. We applied this procedure 10 times for each combination of $U \in \{1, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500, 1000, 2000, 5000\}$ and $C \in \{1, 19, 20, 30, 40, 50, 100, 500\}$, and compute the means of the three execution time measures for each of these combination. Hence 1120 tests were achieved, resulting in 112 results of 3 measures for both commands.

Figure 23 represents the 3 different execution times for *sr* and *sudo* according to the number of items ($U \cdot C$) of the configuration file. While the linear execution time computed previously can be validated, we can also spot some differences between the two programs: a difference of intercept of the lines for both system and user times, that is added up in the overall real time, and a difference of slope for the system time that seems to be present in the overall real time.

In order to confirm these observations, we applied a linear regression for each program regarding the overall real execution time. Our results show a constant difference of around 900ms in execution time between *sr* and *sudo*, shared between kernel and user mode. Our main hypothesis to explain such constant difference resides in the fact *sr* has to load heavy libraries memory at launch that *sudo* do not use, especially *libxml*, used to parse XML files. While this difference is not of importance in a terminal mode where each of the commands are executed by a human operator one after the other, this overhead of time could significantly decrease the performance of a batch processing in the case the script would rely on the *sr* command.

The difference of slope is near 0.02 $ms \cdot kItems^{-1}$; in other words, the difference of execution time between *sr* and *sudo* increases of 0.02ms each thousand items. Regarding Figure 23, this difference of slope seems to reside in the system operations. However, since we consider this difference not high enough to have any significant effect in a real context of use, we did not pursue our analysis in that.

The visible noise at the beginning of the *sr* curves on Figure 23 is related to the XML configuration file reading operation. The current implementation relies on the *libxml* library that operates an initial set of memory operations (mainly allocations) to parse an XML file. In a time-sharing operating system, such operations may have different time duration regarding to the system state, so it cannot be completely deterministic. This random variation on time can be seen on tests with few items, while it becomes negligible to the rest of the processing as number of items to handle increases. In order to address this issue, we will transpose our proposition in a version using flat files structure, or binary representation as in *SELinux*.

7. Conclusions and Future Works

Our module *RootAsRole* provides new functionality to the Linux community by providing them with a module that allows Linux privileges to be given to users through the assignment of roles. Our module also allows Linux administrators to constrain the use of these privileges to certain programs (e.g., the Apache service or *tcpdump*). In addition, *RootAsRole* allows administrators to assign roles to sets of users through the group concept. Our *sr* command is more secure than *sudo/su* commands because it comes only with two capabilities which are *cap_setpcap*

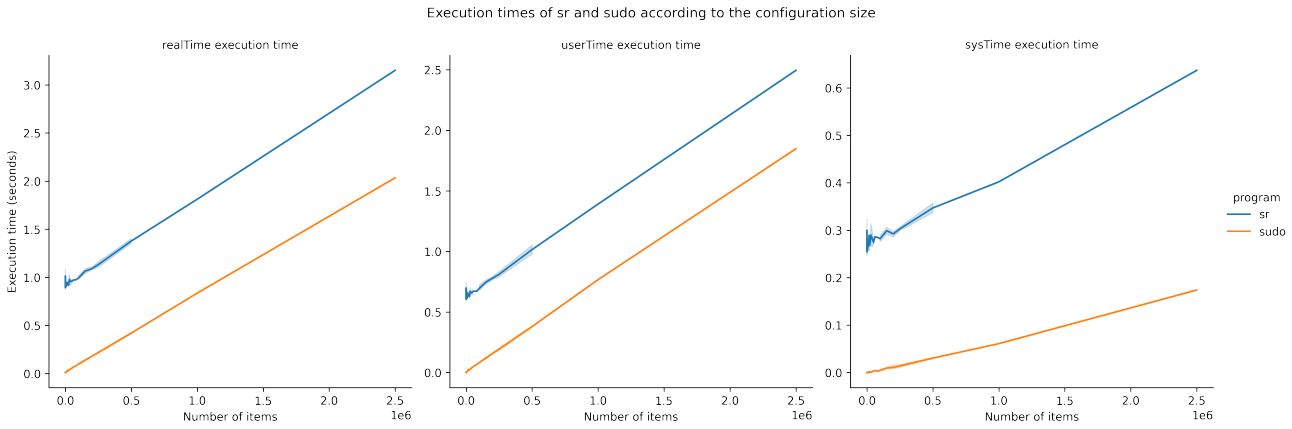

Figure 23. Execution times of *sudo* and *sr*, depending of the configuration size.

and *cap_setfcap*, while *sudo/su* commands need the full list of Linux administrative privileges.

However we need to conduct a user study to analyze the usability issues of our module. We need to test the usability regarding the configuration of *RootAsRole* and the invocation of roles by users. But before conducting such a study, we need to add more tools. Firstly, we would like to add a GUI for setting the configuration policy. Secondly, we will work on defining default policies for different distributions of Linux. Specifically, we will check the list of distributed tools and provide a pre-configured policy for these tools. Thirdly, we would like to integrate the code of *capable* inside *sr*, so that the policy will be configured automatically whenever an administrative user executes a command with *sr*. This will be the case for most personal computer users. Because some programs allow their resources to be listed in their arguments (e.g., the port number for an Apache service or the network interface for *tcpdump*) then administrators can exploit this to limit the use of privileges on resources. For example, an administrator can limit the *cap_net_bind_service* privilege to port 80 by defining a new configuration file for the Apache service and editing the CapabilityRole.xml file so that the user can only start the Apache service with this new configuration file. However, not all programs allow their resources to be listed, and it will not work where users are assigned privileges to run their own programs. As a consequence, additional work is needed to implement these sorts of constraints. In future, we would like to build a kernel or LSM that can intercept all requests for Linux's resources and check whether these are authorized or not, according to the central policy file.

In addition, we would like to extend our module to handle more complex scenarios. For example, giving administrators the possibility to limit the use of privileges not only in terms of programs and resources but also in terms of additional contextual information such as time and location. We would also like to introduce additional RBAC features such as role hierarchies and separation of duties [20].

Finally, we would like to add the logic of *RootAsRole* module into *sudo/su* commands so that users can restrict the usages of administrative privileges when they are using *sudo/su* commands. We would like to investigate the possibility of detecting the needed capabilities and to add them to the policy in a semi-automatic way.

Regarding the Linux capability model, we believe it has some limitations. For example, the capabilities do not have the same level of granularity e.g., *cap_kill*, only permits the killing of other processes, whereas *cap_sys_admin* can be used to handle many different features of the kernel such as administration operations (mount, quotactl, etc.), syslog operations, extended attributes operations, and many others. As pointed out by Michael Kerrisk [8] this problem comes from the fact that Linux does not have a central authority that determines how capabilities should be linked to the kernel features. We believe that this issue constitutes a significant obstacle, but this problem does not affect only *RootAsRole* but all other LSM. Further research is needed to handle this issue because kernel developers define the access controls on kernel resources in an ad-hoc manner [22].

References

[1] Serge E. Hallyn, Andrew G. Morgan, "Linux capabilities: making them work", The Linux Symposium, Ottawa, ON, Canada, 2008, https://www.kernel.org/doc/ols/2008/ols2008v1-pages-163-172.pdf.

[2] "Extended attributes: the good, the not so good, the bad", 2014, https://www.lesbonscomptes.com/pages/extattrs.html, accessed: 29/08/2022.

[3] "LD.SO(8)- Linux manual page", http://man7.org/linux/man-pages/man8/ld.so.8.html, accessed: 29/08/2022.

[4] "errno(3) — Linux manual page", https://man7.org/linux/man-pages/man3/errno.3.html, accessed: 28/08/2022.

[5] "Example code of Python HTTP Server", https://docs.python.org/2/library/simplehttpserver.html, accessed: 29/08/2022.

[6] Karl Fischer, "Intercepting / Hooking function calls to shared C libraries", 2015, https://fishi.devtail.io/weblog/2015/01/25/intercepting-hooking-function-calls-shared-c-libraries/, accessed: 29/05/2022.

[7] Ahmad S. Wazan, David W. Chadwick, Rémi Venant, Eddie Billoir, Romain Laborde, "Code source of RootAsRole module", https://github.com/SamerW/RootAsRole, accessed: 29/08/2022.

[8] Micheal Kerrisk, “CAP_SYS_ADMIN: the new root”, 2012, https://lwn.net/Articles/486306/, accessed: 29/08/2022.

[9] Julianne F. Haugh, Marek Michałkiewicz, Tomasz Kłoczko, Nicolas François, “passwd code source”, https://github.com/shadow-maint/shadow/blob/master/src/passwd.c, accessed: 29/08/2022.

[10] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman, “Linux Security Modules: General Security Support for the Linux Kernel”,11th USENIX Security Symposium (USENIX Security 02),2002, https://www.usenix.org/conference/11th-usenix-security-symposium/linux-security-modules-general-security-support-linux.

[11] Michael Kerrisk, “The Linux Programming interface”, ISBN 159327291X, No Starch Press, October 1 2010.

[12] “capabilities(7) — Linux manual page”, http://man7.org/linux/man-pages/man7/capabilities.7.html, accessed: 29/08/2022.

[13] “eBPF manual page Linux”, http://man7.org/linux/man-pages/man2/bpf.2.html, accessed: 29/08/2022.

[14] “Kernel Probes (Kprobe)”, https://www.kernel.org/doc/Documentation/kprobes.txt, accessed: 29/08/2022.

[15] “Tracepoints”, https://www.kernel.org/doc/Documentation/trace/tracepoints.txt, accessed: 29/08/2022.

[16] “Ftrace”, https://www.kernel.org/doc/Documentation/trace/ftrace.txt, accessed: 29/05/2022.

[17] Matt Fleming, “A thorough introduction to eBPF”, 2017, https://lwn.net/Articles/740157/, accessed: 29/08/2022.

[18] “pid_namespaces(7) — Linux manual page”, http://man7.org/linux/man-pages/man7/pid_namespaces.7.html, accessed: 29/08/2022.

[19] Francesco Pira, “Getting started with AppArmor”, 2016, https://www.slideshare.net/pirafrank/getting-started-with-apparmor, accessed: 29/08/2022.

[20] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, Charles E. Youman, “Role-Based Access Control Models”, 1996, Computer 29, 38–47, doi:10.1109/2.485845

[21] “sudo vulnerability CVE-2019–14287”, 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-14287, accessed: 29/08/2022.

[22] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, Ruowen Wang, “Pex: A permission check analysis framework for Linux kernel”, 2019, Proceedings of the 28th USENIX Conference on Security Symposium, pp. 1205-1220.

[23] Qixu Wang, Dajiang Chen, Ning Zhang, Zhen Qin, Zhiguang Qin, “LACS: A Lightweight Label-Based Access Control Scheme in IoT-Based 5G Caching Context,” in IEEE Access, vol. 5, pp. 4018-4027, 2017, doi: 10.1109/ACCESS.2017.2678510.

[24] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, Martin Gogolla, “Analyzing and Managing Role-Based Access Control Policies,” in IEEE Transactions on Knowledge and Data Engineering, vol. 20, no. 7, pp. 924-939, July 2008, doi: 10.1109/TKDE.2008.28.

[25] Ahmad S. Wazan, David W. Chadwick, Rémi Venant, Romain Laborde, Abdelmalek Benzekri, “RootAsRole: Towards a Secure Alternative to sudo/su Commands for Home Users and SME Administrators”, 2021, In: Jøsang A., Futcher L., Hagen J. (eds) ICT Systems Security and Privacy Protection. SEC 2021. IFIP Advances in Information and Communication Technology, vol 625. Springer, Cham, doi: 10.1007/978-3-030-78120-0_13.

[26] “SUSE product documentation - Building profiles from the command line”, https://documentation.suse.com/sles/15-SP4/html/SLES-all/cha-apparmor-commandline.html#sec-apparmor-commandline-profiling-systemic, accessed: 29/08/2022.

[27] “Apache2-common abstraction file”, https://gitlab.com/apparmor/apparmor/-/blob/master/profiles/apparmor.d/abstractions/apache2-common, accessed: 29/08/2022

[28] David Howells, “Cred structure”, https://github.com/torvalds/linux/blob/master/include/linux/cred.h, lines 128-132, accessed: 29/08/2022.

[29] “AppArmor LSM hooks”, https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/security/apparmor/lsm.c, lines 139-151, accessed: 31/08/2022.

[30] “code source of cap_capable() function”, https://github.com/torvalds/linux/blob/master/security/commoncap.c, lines 66-99, accessed: 31/08/2022.

[31] Andy Lutomirski, “capabilities: Ambient capabilities”, March 2015, https://lwn.net/Articles/636533/, accessed: 31/08/2022.

Appendix
Flow charts of sr and sr_aux

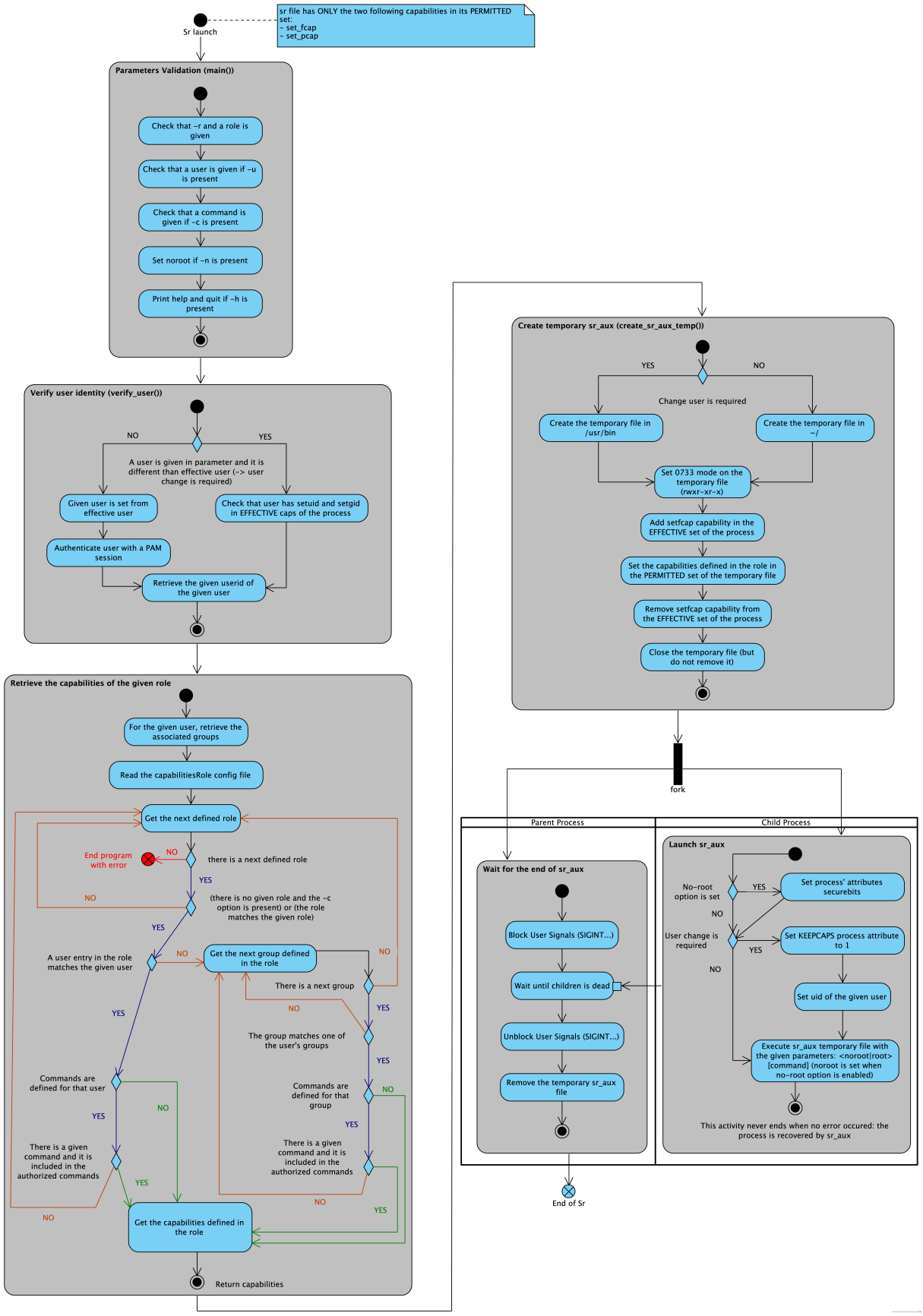

Figure 24. Flowchart for the sr command.

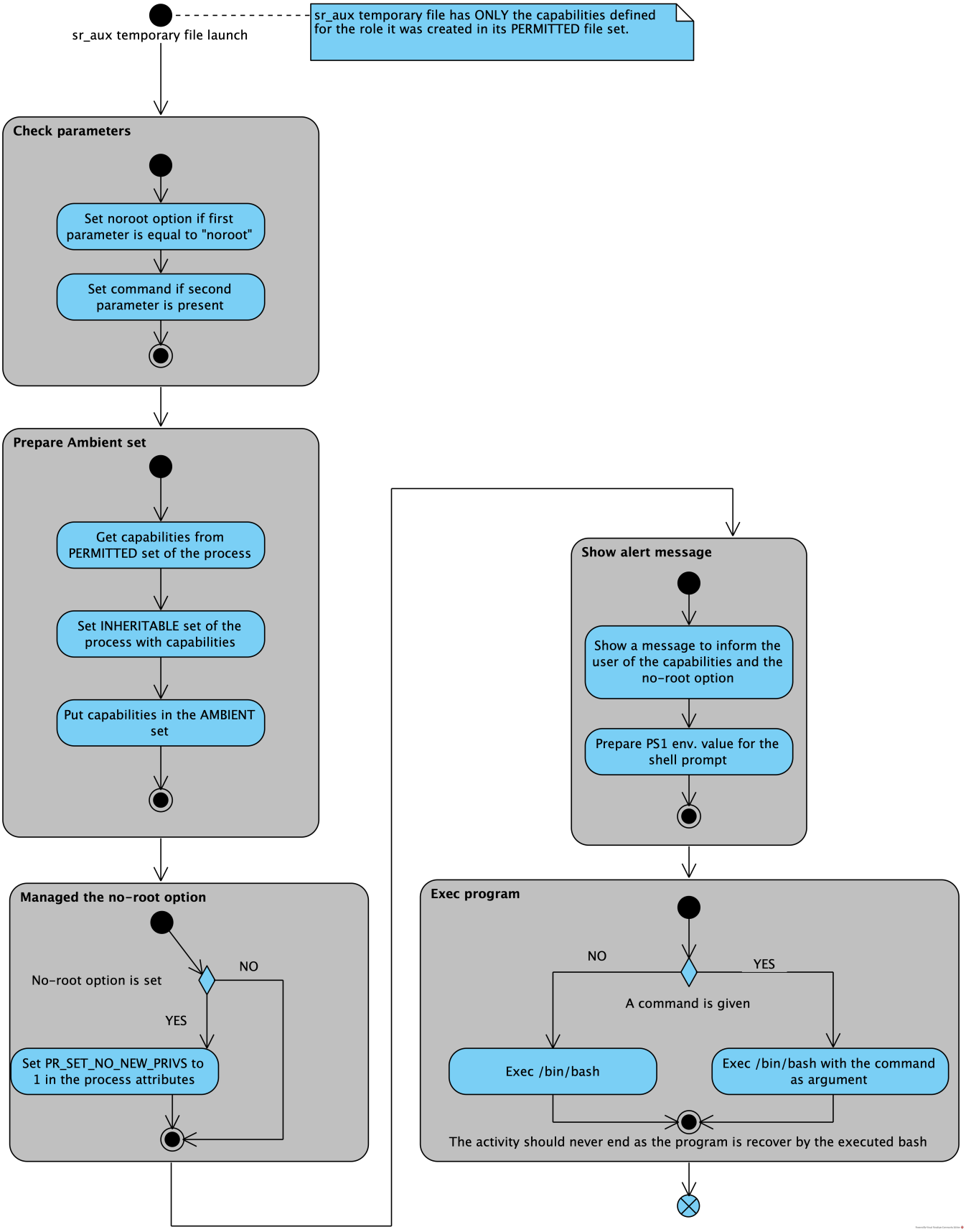

Figure 25. Flowchart for the sr_aux.