# RootAsRole: Towards a Secure Alternative to sudo/su Commands for Home Users and SME Administrators

Ahmad Samer Wazan[1,2], David W Chadwick[3], Remi Venant[4], Romain Laborde[2], and Abdelmalek Benzekri[2]

[1] Zayed University, UAE
[2] Paul Sabatier University, France
[3] Kent University, UK
[4] LeMans University, France

**Abstract.** The typical way to run an administrative task on Linux is to execute it in the context of a super user. This breaks the principle of least privilege on access control. Other solutions, such as SELinux and AppArmor, are available but complex to use. In this paper, a new Linux module, named RootAsRole, is proposed to allow users to fine-grained control the privileges they grant to Linux commands as capabilities. It adopts a role-based access control (RBAC) [14], in which administrators can define a set of roles and the capabilities that are assigned to them. Administrators can then define the rules controlling what roles users or groups can assign to themselves. Each time a Linux user wants to execute a program that necessitates one or more capabilities, (s)he should assign the role to him/herself that contains the needed capabilities, providing there is a rule that allows it. A pilot implementation on Linux systems is illustrated in detail.

**Keywords:** sudo/su commands · 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 super user or root, whose id value is 0. The initial administration model was very simple 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 whilst Effective ID is used to determine 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 [7]. Consequently, any regular user who wants to achieve an administrative task on Linux must change his *euid* or *egid* to 0.

This models creates a problem because all programs executed in the context of the super user can in fact 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, recently a vulnerability has been discvored in sudo program (setuid bit set) that allows any non privileged user to become root [11]. Another example, an administrator who adds to the *sudo* configuration file the possibility for a user test to run the command find could give the user test root access to the whole system, if he was not aware of the existence of the exec argument in the find command. This argument allows the test user to execute arbitrary commands on the system (see figure 1). In addition, the simple administration model can also cause disastrous problems on the system when users commit some fatal error such as "*rm -rf /*" which allows the user to remove the whole files system (see figure 2).



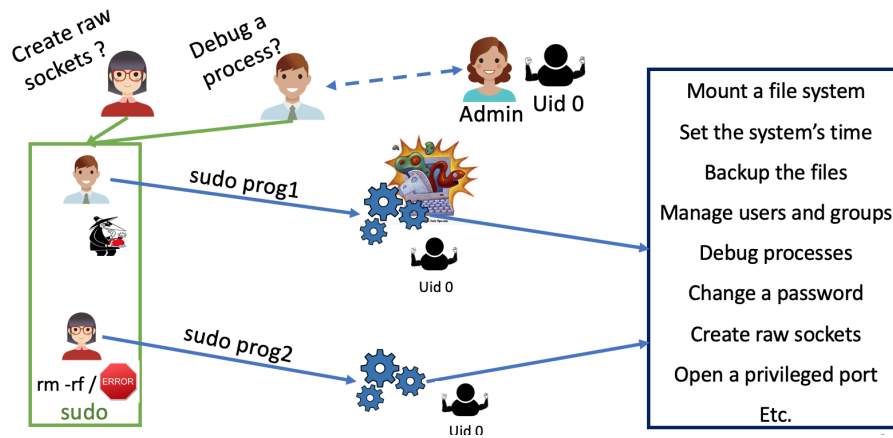**Fig. 1.** Privilege escalation with Find command.



**Fig. 2.** sudo command.

As a consequence, a POSIX draft (POSIX draft 1003.1e) has been proposed in order to create special permissions called Linux capabilities[5]. Although the POSIX draft has been withdrawn, it has been integrated into the kernel of Linux since 1998. We use the term administrative privileges to refer to Linux capabilities. 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, system administrators don't 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 doesn't provide a tool that permits Linux users to know the capabilities that a program needs in order to run successfully (problem 3). Linux kernels come with an emulation mode that allows any process whose *euid* is 0 to have the full list of Linux capabilities, emulating in this way the original administration model. As a consequence, the majority of Linux users still use *su* and *sudo* commands to run privileged programs that obtain the full list of capabilities from the kernel because their *euid* is equal to 0.
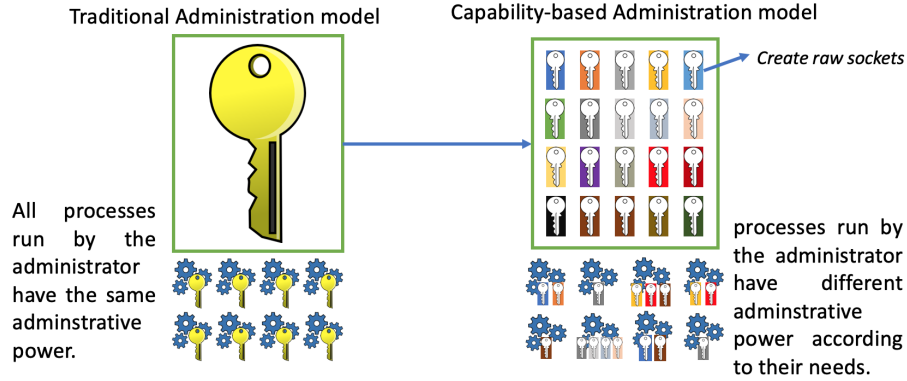


**Fig. 3.** Traditional based processes VS capability based processes.

Thus, the changes that have been made to the Linux kernel since 1998 haven't helped to concretely provide a more secure administrative model. In 2015, the kernel of Linux was modified in order to address the technical problems related

---

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

to the storing of capabilities in the extended attributes of executable files (i.e. problem 1). In addition, this new feature added to the kernel makes it possible to extend the capability module by adding on the top of it a policy management tool that can be configured by the administrators of Linux system.

The objective of this paper is to allow administrator to restrict the use of Linux capabilities in their systems by resolving the problem 2, without necessarily having to extend their kernels with complex Linux security modules (LSM) such as SELinux and AppArmor. This can be especially suitable for home users and for SME administrators. 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.

The paper is organised as follows. Section 2 describes Linux capabilities and shows 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 one motivating scenario. In section 5 we conclude with the limitations of our proposal and our future work.

## 2    Linux Capabilities

Starting with Kernel 2.2, Linux divides the traditional power of the superuser into smaller distinct units called capabilities [8]. 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. 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 rules to be overridden. A complete explanation of the Linux capabilities can be found in the Linux man [8].

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 4). Since the reserved 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.

**Fig. 4.** Values of a shell run by root (cat /proc/$$/status).

Effective contains the set of capabilities that are currently used by the process. textitcap_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 later).

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.

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 don't use the extended attributes correctly. For example, the *mv* command preserves the extended attributes by default. However, the administrator doesn't get a warning message when moving files onto a file system that doesn't support the extended attributes, so they are lost. On the other hand, the cp command doesn't 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 don't 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, or when a program calling a system binary is suddenly halted;
- the Linux kernel doesn't 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.

Regarding the Linux capability model, we believe it has some limitations. As pointed out by Michael kerrisk [6] Linux doesn't 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 doesn't affect only RootAsRole but all other LSM modules. Further research is needed to handle this issue because kernel developers define the access controls on kernel resources in an ad-hoc manner [12].

## 3    Related Works

Some existing Linux security modules can be used to restrict the distribution of Linux capabilities to programs, such as SELinux, AppArmor and Grsecurity. However, these modules allow not only the capabilities' distribution to Linux programs to be managed, but also mandatory access control (MAC) rules to be defined. SELinux implements a label based access control (LBAC) model [13] and uses the extended attributes to store label values. As a consequence, SELinux suffers from the same problems that we mentioned earlier with regards to the extended attributes. In addition, SELinux is very hard to manage even by experienced administrators due to the difficulties with regards to the management of the LBAC system and the MAC policy. Thus, SELinux is not suitable for home users nor for SME administrators.

AppArmor presents an interesting approach because it uses pathname enforcement instead of labels. The syntax for writing AppArmor profiles is much easier than SELinux. AppArmor also provides a tool that automatically generates profiles for programs. The user runs the automation tool and his program at the same time and the tool then asks for the user/Administrator to approve the addition of each generated rule to the profile. Unfortunately, this learning mode fails very often to create automated profiles [9]. Grsecurity adopts like AppArmor a pathname enforcement approach. Grsecurity implements Role Based

Access Control (RBAC) model [10] 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 allows also to determine the list of Linux capabilities that any programs have the authorization to take. However, Grsecurity is not only a LSM module, it is presented as an extensive security enhancements to the Linux kernel because it proposes a set of protection measures against a set of well known security threats.

Theoretically, Linux home users and administrators can use these modules to prevent the privilege escalation problem. However, it is known that the MAC system is disabled on the majority of Linux systems. When activated, the default policy is only applied to the most high risk applications (such as network applications) while the other applications are left unconstrained. Thus, these MAC modules are useless to prevent the root privilege escalation problem because any unconstrained process that is run by *sudo/su* can easily deactivate any of these MAC modules or even remove them. In addition, many home and SME administrators may not need to extend their kernels with complex LSM modules but they may wish to restrict the distribution of Linux privileges without having to modify their kernels. Today, there is no solution to this type of users' need. As a consequence, most Linux users continue to use the *sudo/su* command to manage their own Linux systems because of the complexity of the MAC modules. Our objective is to provide a more usable module where the MAC rules are not needed or where they are only applied partially on the Linux system. This is the case for most home users and SME administrators.

## 4   RootAsRole Module

The pam_cap module is the only module that can be used by administrators to define how capabilities can be distributed to Linux users for systems where MAC rules are not implemented. 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.

In order to solve these problems, we have created the RootAsRole module. Roles are defined by an administrator in a central configuration file, and each role is assigned a set of capabilities. RootAsRole allows Linux users to use the *sr* command to assign roles to themselves. The *sr* command provides options that allow users to know what roles are currently assigned to themselves 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. Administrators define the roles and the rules that allow users or groups to assign the roles to themselves. Each time a Linux user wants to execute a program that necessitates one or more capabilities, (s)he should assign the role to him/herself that contains the needed capabilities, providing there is a rule that allows it.
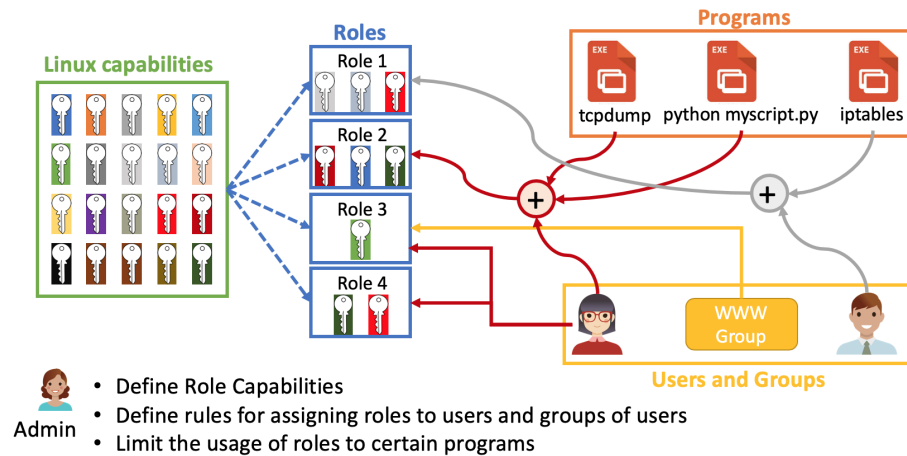
**Fig. 5.** Role definition in RootAsRole module.

The RootAsRole module comes with a configuration file called capability-role.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, the following configuration file defines a role (role1) that has two capabilities (cap_net_raw and cap_net_admin). Users *guillaume* and *remi* can be assigned this role as can be the users of group *adm*. However, *remi* can only be assigned this role when he is executing the *tcpdump* command. The other users 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).

This configuration file 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.
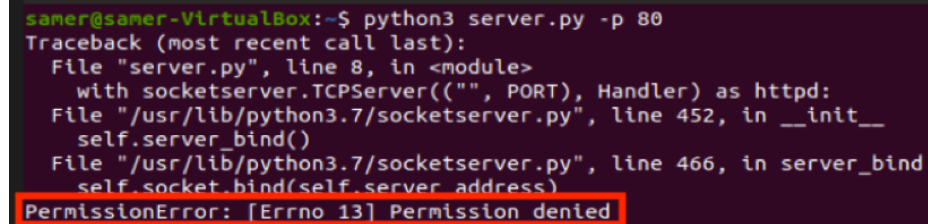
When the administrator lists a program under a user, the user can only run this program with the capabilities. In some cases, a conflict may exist between the programs defined at the user level and those defined at the group level. For example, an administrator may restrict a user to execute only one program, but at the same time he authorises the user's group to execute any program. In these situations, we 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*.The RootAsRole module is released here [5] under GPL-3.0 License.

```xml
1   <configuration xmlns="http://mycapconf.xml">
2       <roles>
3           <role><!-- a role with 2 users, whose 1 has command restrictions -->
4               <name>role1</name>
5               <capabilities>
6                   <capability>cap_net_raw</capability>
7                   <capability>cap_net_admin</capability>
8               </capabilities>
9               <users>
10                  <user>
11                      <name>guillaume</name>
12                  </user>
13                  <user>
14                      <name>remi</name>
15                      <commands>
16                          <command>
17                              <program>/usr/sbin/tcpdump</program>
18                              <args>
19                                  <arg>-i eth0</arg>
20                              </args>
21                          </command>
22                      </commands>
23                  </user>
24              </users>
25              <groups>
26                  <group>
27                      <name>adm</name>
28                  </group>
29              </groups>
30          </role>
31  </configuration>
32
```

**Fig. 6.** Example Definition of a role in the configuration file.

## 5    Motivation Scenario

We will demonstrate the use of our RootAsRole module through one scenario. This scenario shows the advantage of our module with regards the existing tools that are used today to run administrative tasks. *Scenario* A user contacts his administrator to ask for a privilege that allows him to run a HTTP server that he developed using Python. His 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. *Implementation* To demonstrate the implementation of this scenario, we selected the following python script, server.py, which can be used to run a HTTP server [4]. When executing the server.py script without any privileges, we get the '*permission denied*' error message (Figure 7).



**Fig. 7.** Run the Python HTTP server by a normal user.

The script requires the capability cap_net_bind_service 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 8 shows that process 5402 has the full set of Permitted, Effective and Bounding capabilities as these values correspond to the first 38 bits being set to 1.

When an administrator uses the *setcap* command to assign the cap_net_bind_service capability to the python interpreter (Figure 9) he creates a security risk because now all other users of the system will be able to run Python scripts with the same privilege.

**Fig. 8.** Values of the capability sets for a process.



**Fig. 9.** Setting the Permitted and Effective capabilities for the Python2.7 interpreter.

The administrator 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 10.



**Fig. 10.** Setting the Inheritable and Effective capabilities for the Python2.7 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 Figure 11

In this case, the user *awazan* can assign himself the role *role1* by using the *sr* command whenever he wants to run his HTTP server python script (server.py) see Figure 12.

As shown in Figure 13, the user *awazan* cannot run another script with the same privilege. Generally, administrators may not want to limit the use of

```xml
<role name="role1">
  <capabilities>
    <capability>cap_net_bind_service</capability>
  </capabilities>
  <users>
    <user name="root"/>
    <user name="awazan">
      <commands>
        <command>python /usr/local/Users/awazan/server.py</command>
      </commands>
    </user>
  </users>
  <groups>
    <group name="adm"/>
    <group name="office"/>
    <group name="secretary">
      <commands>
        <command>/usr/bin/printer</command>
        <command>/usr/bin/other</command>
      </commands>
    </group>
  </groups>
</role>
```

**Fig. 11.** Defining role1 for running the Python HTTP server.

```
[awazan@dane:~$ sr -r role1 -c "python /usr/local/Users/awazan/server.py"
Authentication of awazan...
[Password:
Privileged bash launched with the following capabilities : cap_net_bind_service.
serving at port 80
```

**Fig. 12.** User awazan assigns the role role1 to execute the Python HTTP server.

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 don't completely trust all their users.



**Fig. 13.** user awazan unable to run another Python script.

## 6    Discussion, Limitations and Conclusions

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 home users and/or SMEs 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* commands is more secure than sudo/su commands because it comes only with two capabilities which are cap_setpcap 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 study, we need to add more tools. Firstly, we are currently finalizing the implementation of a tool called *capable* that will allow an administrator to know the set of privileges requested by a program. In addition, we would like to implement a tool that allows system administrators to easily edit our central configuration file e.g. to add roles to the configuration file along with the list of privileges, users, groups and authorized programs. Secondly, we would like to add a GUI for setting the configuration policy. Thirdly, 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.

Finally, 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 [10].

## Acknowledgement

## References

1. J2. 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. Last Access: 28/03/2021.
3. "Linux manual page:ld.so, ld-linux.so - dynamic linker/loader", http://man7.org/linux/man-pages/man8/ld.so.8.html. Last Access: 28/03/2021
4. Example code of Python http Server, https://docs.python.org/2/library/simplehttpserver.html. Last Access: 28/03/2021
5. Code source of RootAsRole module, https://github.com/SamerW/RootAsRole. Last Access: 28/03/2021
6. Micheal Kerrisk, "CAP_SYS_ADMIN: the new root", 2012, https://lwn.net/Articles/486306/. Last Access: 28/03/2021
7. Michael Kerrisk, "the Linux Programming interface", ISBN 159327291X, No Strarch Press, October 1 2010.
8. "Linux capabilities man page", http://man7.org/linux/man-pages/man7/capabilities.7.html. Last Access: 28/03/2021
9. "Getting started with AppArmor", https://www.slideshare.net/pirafrank/getting-started-with-apparmor. Last Access: 28/03/2021
10. Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman, "Role-Based Access Control Models", 1996, Computer 29, 3847. DOI:https://doi.org/10.1109/2.485845
11. "sudo vulnerability CVE-201914287", https://medium.com/@isharaabeythissa/cve-2019-14287-sudo-will-hit-your-root-4df17e6a089b. Last Access: 28/03/2021
12. T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, R. Wang, "Pex: A permission check analysis framework for linux kernel", 2019, Proceedings of the 28th USENIX Conference on Security Symposium, pp. 1205-1220.
13. Q. Wang, D. Chen, N. Zhang, Z. Qin and Z. 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.
14. K. Sohr, M. Drouineaud, G. Ahn and M. 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.