Implementing the Principle of Least Privilege Using Linux Capabilities: Challenges and Perspectives

Eddie Billoir, Romain Laborde, Ahmad Samer Wazan, Yves Rütschlé, Abdelmalek Benzekri

Implementing the Principle of Least Privilege using Linux Capabilities: Challenges and Perspectives

Eddie Billoir
Airbus Protect, IRIT
Toulouse, France
eddie.billoir@airbus.com

Romain Laborde
IRIT, Université Toulouse III Paul Sabatier
Toulouse, France
romain.laborde@irit.fr

Ahmad Samer Wazan
Zayed University
Abu Dhabi
ahmad.wazan@zu.ac.ae

Yves Rütschlé
Airbus Protect
Toulouse, France
yves.rutschle@airbus.com

Abdelmalek Benzekri
IRIT, Université Toulouse III Paul Sabatier
Toulouse, France
abdelmalek.benzekri@irit.fr

***Abstract*—Historically and by default, Linux does not respect the principle of least privilege because it grants all the privileges to administrators to execute their tasks. With the new personal data protection or export control regulations, the principle of least privilege is mandatory and must be applied even for system administrators. The Linux operating system, since version 2.2, divides the privileges associated with the superuser into distinct units called capabilities. Linux capabilities allow coarse-grained access control to restricted system features. The "RootAsRole" project is introduced as a solution for delegating administrative tasks while matching the necessary capabilities. However, limitations in user experience and the mapping of Linux capabilities pose significant obstacles. This paper proposes enhancements to balance usability and the principle of least privilege, emphasizing the need for precise capability definitions. Future work involves enhancing the RootAsRole access control model and addressing the administration access control framework for managing Linux capabilities effectively.**

***Index Terms*—Access Control, Least privilege principle, Linux kernel, Capabilities**

I. Introduction

The Principle of Least Privilege (POLP) is an engineering process that involves understanding users' responsibilities to grant them the absolute minimum permissions required for accomplishing their tasks using computer systems [1]. This principle applies to all users, especially those responsible for system administration, who often possess comprehensive privileges directly or indirectly.

On the one hand, POLP is essential from a security point of view to minimize the potential attack surface and reduce the potential damage in case of a security breach. In addition, it is the cornerstone of modern security models such as the zero-trust security strategy [2], which sets the least privilege as one of its core principles. On the other hand, POLP is also mandatory to comply with regulations related to personal data (e.g., GDPR [3]) or export control [4]. New hybrid usage of IT devices, either Personally Owned/Company Enabled or Corporate-Owned/Personally Enabled, requires fine-grained administrative privileges to prevent unlawful access to personal data. Co-administration of devices within the organization or outsourcing to third parties is another illustration of this need.

POLP can be implemented on Linux at different levels using different mechanisms provided by the operating system (e.g., the POSIX discretionary access control mechanisms, setuid/sudo, Linux capabilities or mandatory access control mechanisms provided by Linux Security Modules). Nevertheless, we demonstrated that implementing POLP correctly at the OS level is still challenging [5].

This paper focuses on Linux capabilities that allow coarse-grained access control to specific kernel tasks [6]. Indeed, Linux divides the privileges associated with the superuser into distinct units called capabilities. Unlike traditional POSIX permissions tied to file access control, these specific permissions are assigned to programs or processes. They enable privileges to be assigned to individual applications or tasks without the need for them to run with root, the superuser account. We explore different approaches to uphold POLP through OS Hardening and at the user level using our RootAsRole project. We present the limitations and how we have designed new solutions to resolve them. Although our investigations focus on specific tools, they pertain to generic issues regarding using Linux capabilities.

The rest of the article is structured as follows. Section II provides an introduction to Linux capabilities. Section III analyses the pros and cons of implementing POLP by hardening the OS. Section IV describes the RootAsRole project and the difficulties of implementing POLP while considering the users' experience. Section V discusses the limitations regarding implementing the least privilege principle with Linux capabilities. Finally, we conclude and present future works.

II. Linux capabilities

Linux Capabilities were initially introduced in the Linux kernel version 2.1 in 1998, with implementation occurring in version 2.2 and offering 32 permission slots. As system scaling has progressed, the implementation has been enhanced to manage 64 permissions and enable complex security policy. In the current version (v6.3), 41 capabilities have been implemented [7]. This mechanism can grant access to kernel features or enable bypassing some security policies. As an

example, CAP_NET_BIND_SERVICE is a capability that is needed to bind a socket on TCP/UDP ports between 1 and 1024, and CAP_DAC_OVERRIDE allows bypassing the DAC file-system access control. Due to the current implementation slot limit, Linux decided that CAP_SYS_ADMIN is an overridden capability that doesn't define any scope; this decision offers flexibility for kernel designers to manage capability definitions.

The kernel maintains for each process 5 sets of capabilities in memory to define the state of the discretionary and mandatory privilege policy. These sets is the mechanism to manipulate privileges for a time-of-use criterion and distribute them across the child processes (discretionary policy). If a program executes another binary, the system authority defines which privileges will be inherited (mandatory policy). By default on Linux, a single mandatory policy is applied, which is intrinsic to the kernel. In the kernel implementation of capability sets and associated behaviours, Linux adopts a vocabulary and employs mechanisms akin to the capability-based access control model mechanisms. While Linux's model is not explicitly described as respecting this model, it shares common elements such as Ambient authority. This suggests that Linux adapts the capability-based security model to fit specific OS needs [8], [9].

Linux Capabilities are primarily used to confine applications in Linux namespaces, such as Docker/Podman or other LXC solutions. The Linux namespaces isolate processes by giving them independent and limited views of system resources, such as process IDs, network interfaces, and file systems, providing a form of process-level virtualization. LXC, Docker and Podman are three solutions for creating and managing isolated environments, mainly using Linux namespaces. Although these products allow you to manipulate Linux capabilities, they do not offer better usability or understanding.

III. The OS hardening approach

Generally, an OS offers multiple generic features that may not be needed for a specific system usage. Implementing POLP may be needed to inhibit the useless features of the entire OS. This process, also known as OS Hardening, involves modifying or configuring an operating system to minimize its features, and hence the impact of potential vulnerability on the entire system. For instance, when the file-system access control is correctly configured, the person in charge of configuring the system may want to prevent the superuser from bypassing this control by removing CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH capabilities. OS hardening should be carefully implemented because it could lead to safety risks due to the discretionary access control properties [16].

Different approaches can be followed to implement this goal. First, this could be achieved by using BPF LSM (Linux Security Modules [14]), which is a framework that allows developers to write security systems on top of the Linux kernel using eBPF (extended Berkeley Packet Filter) programs. eBPF is a technology that enables sandboxed programs to run in the Linux kernel without modifying the kernel source code or loading a kernel module. These programs can be loaded and unloaded at any time by any user with the CAP_BPF capability. It is possible to deny any CAP_DAC_OVERRIDE capability request using such technology. However, any privileged session could unload the eBPF program and regain its privileges because it has them in its privilege sets. We've initiated a repository with a proof-of-concept of this example [15].

Another approach consists in creating a new Linux kernel module (LKM) to remove the CAP_DAC_OVERRIDE capability from every process credential created on the system. This can be done by manipulating the capabilities credentials structure during the 'execve()' process. This prevents processes from retrieving their capabilities without unloading the module with the CAP_SYS_MODULE capability. The advantage of this strategy is that most of the processes will have their capabilities permanently removed. Indeed, modules are loaded by the "(systemd-)udev" process, which is initialized by the init or systemd programs. These program processes do not lose their capabilities and can still be used. We've also created an example of such a kernel module on the same repository as eBPF [15].

```
 # define CAP_FS_MASK     (BIT_ULL(CAP_CHOWN)           \
                         | BIT_ULL(CAP_MKNOD)           \
-                        | BIT_ULL(CAP_DAC_OVERRIDE)    \
-                        | BIT_ULL(CAP_DAC_READ_SEARCH) \
                         | BIT_ULL(CAP_FOWNER)          \
                         | BIT_ULL(CAP_FSETID)          \
                         | BIT_ULL(CAP_MAC_OVERRIDE))
-#define CAP_VALID_MASK  (BIT_ULL(CAP_LAST_CAP+1)-1)
+#define CAP_VALID_MASK  ((BIT_ULL(CAP_LAST_CAP+1)-1) \
+                        & ~(BIT_ULL(CAP_DAC_OVERRIDE) \
+                          | BIT_ULL(CAP_DAC_READ_SEARCH)))
-#define cap_raise(c, flag)  ((c).val |= BIT_ULL(flag))
+#define cap_raise(c, flag)  ((c).val |= (BIT_ULL(flag) & CAP_VALID_MASK))
```

Fig. 1: Lines to patch to remove capabilities in kernel v6.3

Finally, the administrator can also patch the current Linux kernel to change the initial set of capabilities permanently. In this way, the capability doesn't exist on the running kernel system. The main disadvantage of this solution is the need to recompile the patch each time the Linux kernel is updated. Also, the Linux kernel is governed by GPLv2 licensing, so these modifications must comply with the license terms. To perform this concretely, the definitions of pre-processor instruction on file "include/linux/capability.h" needs to be edited. We described these changes in Figure 1.

By hardening the OS, the capabilities of Linux allow the person in charge to configure the system to eliminate security risks but also to increase the safety risks if misconfigured. This person could use more or less permanent approaches to simplify recovery. Nevertheless, this measure does not respect POLP enough. As said earlier, the least privilege is based on understanding users' task needs. OS hardening can only apply the same configuration to all users, i.e., removing Linux capabilities not required by all users. Therefore, another mechanism is needed to apply POLP at the user's level, including administrators.

IV. RootAsRole

We present in this section RootAsRole, a new security mechanism we developed for controlling Linux capabilities at the user level. We describe our goal and the latest improvements we introduced to provide a useful tool that allows fine-grained implementations of POLP. We also highlight open questions.

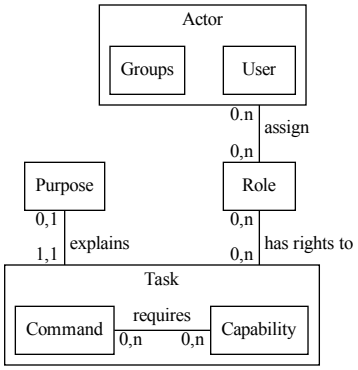

Fig. 2: Design of role configuration model

Today, most Linux distributions propose the *sudo* command to elevate privileges. *sudo* is a tool that allows a system administrator to delegate commands with potentially all root privileges to users [10]. *sudo* includes many other security features, but we will not elaborate on them. However, *sudo* does not handle Linux Capabilities. Since no sophisticated and easy-to-use mechanisms were available to manage capabilities, we developed RootAsRole [12], [13] a new security module to control the Linux capabilities given to applications. In addition, unlike *sudo*, which does not include any access control model, We choose to implement the Role-Based Access Control model (RBAC) that consists in granting (and restricting) access permissions to roles, and then these roles are assigned to users [16]. The RBAC model allows grouping administrative privileges and tasks by roles. We chose this model because it impels the implementation of the POLP since tasks and duties must be explicitly analyzed to identify roles.

To implement least privilege more precisely with RootAsRole, we took a lot of initiatives to resolve design, usability and reliability issues for both users and administrators. To illustrate these conceptual issues, we introduce a tiny web business example. Let's consider user Alice who is in charge of managing an apache2 web server installed on a Linux machine. Consequently, she should be allowed different tasks such as starting/stopping the web server, modifying the web server configuration files, etc. In addition, Alice is a web developer. So, she should be able to add her code to the web server but also use the command *tcpdump* to debug the new web protocol she is implementing. We will develop this example in the rest of this article to exhibit the issues when implementing POPL and how we partially handle them.

```
1  <role name="web_admin">
2     <actors>
3        <user name="alice"/>
4     </actors>
5     <task capabilities="cap_net_bind_service"
6              setuser="apache"
7              setgroups="apache">
8        <command>/usr/local/sbin/apache2ctl start</command>
9        <command>/usr/local/sbin/apache2ctl restart</command>
10       <command>/usr/local/sbin/apache2ctl reload </command>
11       <purpose>Manage the apache2 service</purpose>
12    </task>
13    ...
14 </role>
15 <role name="web_dev">
16    <actors>
17       <user name="alice"/>
18       <group names="softteam"/>
19    </actors>
20    <task capabilities="cap_net_raw,cap_net_admin">
21       <command>/usr/bin/tcpdump</command>
22       <purpose>Debug HTTP responses</purpose>
23    </task>
24    ...
25 </role>
26 ...
```

Fig. 3: A Sample RootAsRole policy for our webserver example

A. Administrative issues

1) Making Linux capabilities and POSIX DAC policies consistent: We developed a language that allows the administrators to specify which users can use which command with which capabilities. This language extends the RBAC model to include capabilities in the permissions assigned to roles. The implementation of our model is described in Figure 2. In this model, actors which can be Linux users and/or Linux groups are assigned to roles. Roles are assigned to permissions to perform tasks which are sets of commands and allowed Linux capabilities. We also require the administrator to explicitly state the purpose of the permission assignment in a human-readable format to enhance the maintainability of the policies. This new model improves the initial one.

Figure 3 is the RootAsRole policy of the use case described in the project presentation section. This policy includes two roles: `web_admin` and `web_dev`. The `<actors>` element inside the role definition represents the user assignment relation. Here, Alice has been assigned to both roles (see lines 3 and 15). It is also possible to assign a Linux group to a role like in line 16. Administrators can specify the tasks assigned roles by including them in `<task>` elements. Each task lists a set of commands and the associated permitted Linux capabilities. For instance, role `web_admin` can use `CAP_NET_BIND_SERVICE` (i.e., bind a port less than 1024) for starting, restarting and reloading the apache2 web server. This task is related to managing the apache2 service as described in line 11.

In order to assign tasks to their users, administrators need to manipulate the entire credentials context and environment variable sessions. For example, Alice may need to change her effective user to a dedicated system user (e.g., user `apache`) when configuring the apache2 web server to be consistent with the DAC policy applied on the file system. In the previous

version of our tool, RootAsRole managed the Linux capabilities feature exclusively, resulting in inconsistencies between RootAsRole policies and DAC policies. So we implemented the effective user/groups change for a task. The tool also manages an environment variable policy that applies a whitelist, a filter list (remove if unsafe) and a define/replace list. Other variables are removed from the created session. The "remove if unsafe" notion is arbitrary according to various vulnerabilities found on shells. This policy is similar to the *sudo* tool. We noticed that *sudo* enables an administrator to choose different algorithms for managing the environment variable. But most of them are not recommended to use because of their potential vulnerabilities e.g. CVE-2014-9680, CVE-2014-10070, CVE-2014-0106, to name but a few. This explains why we chose to implement only the default one that is currently considered safe.

2) Capabilities are unknown by administrators:

```
alice@webserver:~$ capable -c "/usr/sbin/tcpdump"
tcpdump: enp1s0: You don't have permission to capture on that
device
(socket: Operation not permitted)

Here are all the capabilities intercepted for this program :
cap_net_admin, cap_net_raw
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
```

Fig. 4: Capable command output for tcpdump command

We noticed that Linux capabilities are poorly understood, except for kernel developers [2]. Indeed, the Linux usage manual describes capabilities at high level but it does not explain their precise scope, nor their exact purpose for each system calls. Therefore, it was extremely complicated to configure a policy. To help administrators in this task, we developed a tool called *capable* that detects the requested capabilities for a specific command. For example, Alice can use *capable* to determine which capability is needed for the *tcpdump* tool. Here *tcpdump* requires network capabilities, which are identified by our tool and displayed in figure 4.

Capable uses eBPF technology to hook into the capability verification method of the kernel to collect what capability is requested for all processes. It then sets up an unprivileged namespace for the analyzed application before running it. When filtering with the namespace identifier, eBPF can identify the privileges requested for the program. Any requested privileges are printed to the console when the program exits.

In the previous version of RootAsRole, during our experiments, we observed a recurring occurrence of the `CAP_SYS_ADMIN` capability (this capability grants many privileges) being requested. This was attributed to the systematic and repeated call of the `cap_vm_enough_memory()` hook during the memory allocation and process creation stages; one such call stack trace is detailed in Figure 5 using the *bcc* tool [18]. Fortunately, the above problem has since been fixed in the Linux kernel. Nevertheless, this problem

```
TIME      UID    PID    TID    COMM       CAP  NAME
22:58:49  1000   27408  27408  capable    21   CAP_SYS_ADMIN
        cap_capable+0x1 [kernel]
        cap_vm_enough_memory+0x2b [kernel]
        security_vm_enough_memory_mm+0x34 [kernel]
        mmap_region+0x147 [kernel]
        do_mmap+0x38d [kernel]
        vm_mmap_pgoff+0xd2 [kernel]
        elf_map+0x58 [kernel]
        load_elf_binary+0x4cd [kernel]
        search_binary_handler+0x90 [kernel]
        __do_execve_file.isra.36+0x5b1 [kernel]
        __x64_sys_execve+0x34 [kernel]
```

Fig. 5: *bcc* Kernel stack trace describing the problematic capacity demand when using capable tool in kernel version v4.x

highlighted the possibility that certain capabilities might not be requested at the appropriate stage in the kernel algorithm. To ensure the reliability of our tool and identify any misplaced capabilities within the kernel, we developed a straightforward Clang plugin that utilizes Abstract Syntax Tree (AST) analysis. The Clang AST represents the structure and semantics of C/C++ code, serving the purpose of analysis, transformation, and code generation. Our plugin operates on the assumption that capabilities should be the final condition checked, i.e., practically function `capable()` which is called to check capabilities should be placed at the end of the condition of the if statement that implements the access control mechanism. This is a common best practice followed by kernel developers. Indeed, the function `capable()` verifies namespaces capabilities and checks the `security_capable()` LSM hook that could potentially affect the performance of the kernel. This approach enables us to know when a capability is required because function `capable()` is called when all other access control systems deny access. During our analysis on kernel v6.3, our plugin detected 8 occurrences not complying with the best practice . The plugin source code can be found in a GitHub repository [19]. However, some calls do not exist within a conditional statement, so this work only covers 91% of the total calls (993 calls in kernel v6.3).

B. User usability issue

One of the success factors of command *sudo* is its ease of use for final users who are only required to add *sudo* before the command they want to execute. Command *sudo* allows users to run a command as a specific user other than the default target user. But this feature is rarely used.

```
sr -r web_dev -c /usr/bin/tcpdump
```

Fig. 6: The user command to execute tcpdump in the V1/V2 of RootAsRole

In the previous version, we imposed users to explicitly express the role they want to activate to execute a given command (see Figure 6). If users are assigned to multiple

```
alice@webserver:~$ sr /usr/bin/tcpdump
```

Fig. 7: New version of RootAsRole: Alice doesn't need to specify the role to activate

roles, this requirement impacts the users' experience making our security mechanism hard to use for day-to-day tasks.

Consequently, we improved our tool to make the explicit role specification optional as in Figure 7. However, this raises new questions, for instance, when a user is associated with more than one role allowing to execute the same command but with different capabilities. There are multiple algorithms to find out the matching role for a user and a command input. This issue has been thoroughly studied in the context of firewall rules analysis [21] to resolve shadowing, correlation and redundancy anomalies.

We applied the following criteria for order comparison:

1) Find all the roles that match the user id assignment or the group id, and the command input
2) Within the matching roles, select the one that is the most precise and least privileged such as:
   a) user assignment is more precise than the combination of group assignment
   b) the combination of group assignment is more precise than single group assignment
   c) exact command is more precise than exact command with regular expression
   d) A role granting no capability is less privileged than one granting at least one capability
   e) A role without setuid is less privileged than one has setuid.
   f) if no root is disabled, a role without 'root' setuid is less privileged than a role with 'root' setuid
   g) A role without setgid is less privileged than one with setgid.
   h) A role with a single setgid is less privileged than one with multiple setgid.
   i) if no root is disabled, a role with multiple setgid is less privileged than one with set root gid
   j) if no root is disabled, a role with root setgid is less privileged than one with multiple set gid, particularly using root group

If this algorithm doesn't resolve the conflict, roles are considered equal (i.e., the only difference is environment variables). In such cases, the user must specify the role to be used with the '`-r`' option. Regarding point (d), the choice of least privilege is somewhat arbitrary. In fact, in our search for a partial order operator for our tool, we tried to find a partial order between Linux capabilities but could not find it (see Section V).

V. Limits and Discussions

Our tool called "*capable*", can be used to obtain the Linux capabilities needed to execute a given command in a safe mode. However, it does not guarantee that the granted capabilities are the strict minimum required by the program. This is because different capabilities can provide the same kernel feature. For example, both `CAP_BPF` and `CAP_SYS_ADMIN` can grant privileges to load/unload a BPF module (refer to Figure 8). The question arises: which Linux capability should be assigned? From the principle of the POLP perspective, the answer is the capability that grants the least privileges.

```
static inline bool bpf_capable(void)
{
    return capable(CAP_BPF) || capable(CAP_SYS_ADMIN);
}
```

Fig. 8: Sample of the kernel code in file include/linux/capability.h

To address this issue, we attempted to establish a partial order between capabilities regarding the least privileges. In other words, we aimed to identify the less privileged capability compared to others. Our approach involved examining the restricted sections of the kernel source code and determining the specific capability required for each section.

Detecting a restricted feature that may require one or more capabilities and identifying the least privileged capability among them presented a challenge. By identifying the accessed source code, we could determine if one capability is a subset of another and prove that one capability is less privileged. We explored various approaches to automate this analysis, considering that the output of the analysis may change with each kernel version.

A. The theoretical approach

Theoretically, we could adopt a brute-force approach, exhaustively checking the combinations of privileges in the system. This approach can uncover numerous vulnerabilities in manipulating the Linux kernel. However, the number of combinations grows exponentially with the number of capabilities, resulting in an impossible number of possibilities. Considering that there are 41 capabilities, testing all combinations from 1 individual capability to 41 combined capabilities would require evaluating $41! = 3.345^{49}$ possibilities, which is impractical. Even assuming that some privileges are unrelated and reducing the number of combinations, such an analysis remains infeasible due to the scale of the Linux kernel system calls variety. Each combination attempt involves numerous complex operations on the system, such as a system reboot, which can take several seconds and complicate the implementation of any solution.

B. The pragmatic approach

An alternative approach involves analyzing the kernel source code and identifying all system conditions that require privileges. Within the kernel, a function called '`capable()`' exists, enabling testing if any current calling process is granted for a capability. By analyzing the Linux kernel code, we could establish direct connections between privileges, allowing us to

establish an ordering relationship among them. This ordering relationship could help to determine which privilege provides access to a greater range of functionalities than others. Initially, we considered utilizing the *yacc* and *lex* tools and using the ANSI C standard for source code analysis. However, these standards do not specify the preprocessor. So we faced an error from these tools.

```
if\s*(\((?>[^()]+|(?1))*\))\s*
({(?>[^{}]+|(?2))*}|(?R)|[^{};]*;)(?:\s|\n)*
(?:else\s*({(?>[^{}]+|(?2))*}|(?R)|[^;]*;))?
```

Fig. 9: PRCE2 expression match if C language statement

To overcome *yacc* and *lex* limit efficiently, we explored the use of a PCRE2 regular expression to identify "if" conditions in the source code (see Figure 9). PCRE2 (Perl Compatible Regular Expressions 2) is a versatile library that enables advanced pattern matching using regular expressions. The regular expression provided in Figure 9 captures three groups: the "condition," the "then" block, and the "else" block if present. We can recursively apply this expression to each group to identify nested conditions within "if" statements. This approach is advantageous because the parser does not crash when encountering preprocessor syntax. Using this method, we could identify most of the '`capable()`' usage functions (903 out of 993). However, there remain 90 capability requests that were not matched by our plugin. These unmatched calls indicate a limitation in our approach, but these few calls make it possible to identify them manually. Another challenge with this method is still the preprocessor. We found some calls to '`capable()`' are done within preprocessor functions. These preprocessor functions define and replicate the definition of functions only after precompilation. Thus, we are unable to identify a large number of calls due to this technical issue.

Another envisioned approach was utilizing the compiler to perform Abstract Syntax Tree (AST) analysis. This approach would enable us to create a call graph and determine the capabilities implied by the kernel. However, due to time constraints, we have yet to explore and implement this approach. This research is ongoing, and the initial AST plugin is available on GitHub [19].

C. The reverse engineering approach

We also explored the possibility of finding compiled symbols in the kernel binary. Specifically, we aimed to locate the symbol for '`capable()`'. Although we were able to manually identify the symbol using the Ghidra tool, finding the symbol automatically is more complex than other automated methods. The binary search for the symbol would differ for each kernel version, making it challenging to automate this process.

In conclusion, we have yet to discover a straightforward method for mapping Linux capabilities within the kernel. Consequently, a more practical approach would be implementing a development process for kernel developers to document the kernel precisely. We acknowledge that this process is arduous and requires meticulous development, demanding significant time for verification.

VI. Related Work

In 2005, Krohn and team [25] presented "Asnix", an imaginary operating system with an object-capability security model [26]. However, their analysis did not explore Linux Capabilities due to challenges faced by Linux for years. Subsequent developments in Linux have resolved these issues, enabling further exploration of Linux capabilities in subsequent research.

Hallyn et al. [27] presented the use of Linux Capabilities as a working solution for addressing POLP challenges. While this work demonstrated the effectiveness of the Linux capability model, it also highlighted the ongoing difficulties in its practical implementation. Consequently, Linux capabilities have primarily been utilized for enhancing container isolation and fixing vulnerabilities.

In our previous works [12], [13], we introduced a novel software that utilizes Linux Capabilities to implement Role-Based Access Control. These papers proved the potential of Linux Capabilities for daily use but still need to be improved with security and usability efforts.

VII. Conclusion

POLP is a well-known and established security principle. We studied different approaches to implement the POLP using Linux capabilities and demonstrated this is actually a tricky problem. OS hardening applies to each user, so it cannot be considered a fine-grained approach. We developed the "RootAsRole" project that enables administrators to delegate administrative tasks and grant the matching needed capability. However, we highlighted limitations, especially regarding the user experience of both administrators and final users. Finding a good tradeoff between usability and POLP is complex. We proposed in this article enhancements in this perspective.

One of the main challenges is defining precisely the relations between capabilities. Indeed, how can administrators apply POLP when the privileges associated with Linux capabilities are not precisely defined? We haven't found any automatic way to map Linux capabilities into a partial order set by analysing the kernel. This work is still ongoing. We believe the best solution is by using the map created by

For future work, we are working on improving the RootAsRole access control model to allow dynamic security to be enforced, like in [24]. Indeed, many Linux capabilities enable bypassing permission checks. Thus, Linux capabilities require a proper administration access control model to manage administrative privileges.

VIII. Aknowledgements

We are grateful to Raoul Guiazon for his assistance and feedback.

REFERENCES

[1] "Least Privilege — CISA." https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege.

[2] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero Trust Architecture" National Institute of Standards and Technology, Aug. 2020. doi: 10.6028/NIST.SP.800-207.

[3] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance), vol. 119. 2016. http://data.europa.eu/eli/reg/2016/679/oj/eng

[4] Regulation (EU) 2021/821 of the European Parliament and of the Council of May 2021 setting up a Union regime for the control of exports, brokering, technical assistance, transit and transfer of dual-use items (recast). 2022. http://data.europa.eu/eli/reg/2021/821/2022-01-07/eng

[5] E. Billoir, R. Laborde, A. S. Wazan, Y. Rütschlé, and A. Benzekri, 'Est-il difficile de respecter le principe de moindre privilège sur Linux?', presented at the Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information, May 2023. https://hal.science/hal-04103463

[6] "capabilities(7) - Linux manual page." https://man7.org/linux/man-pages/man7/capabilities.7.html.

[7] "linux/capability.h at v6.3 — torvalds/linux — GitHub." https://GitHub.com/torvalds/linux/blob/v6.3/include/uapi/linux/capability.h

[8] K. Leffew, "What is Capability-based Security?" Medium, Nov. 07, 2019. https://medium.com/@kleffew/what-is-capability-based-security-227c6e5483a5.

[9] Miller, Mark & Yee, Ka-ping & Shapiro, Jonathan. "Capability Myths Demolished", 2003.

[10] "Sudo," *sudo*. https://www.sudo.ws/.

[11] SamerW, "RootAsRole : a secure alternative to sudo/su on Linux systems". May 21, 2023. https://github.com/SamerW/RootAsRole

[12] A. S. Wazan et al., "RootAsRole: a security module to manage the administrative privileges for Linux," Computers & Security, p. 102983, 2022, doi: https://doi.org/10.1016/j.cose.2022.102983.

[13] A. S. Wazan, D. W. Chadwick, R. Venant, R. Laborde, and A. Benzekri, "RootAsRole: Towards a Secure Alternative to *sudo*/su Commands for Home Users and SME Administrators," in ICT Systems Security and Privacy Protection, Cham, 2021, pp. 196–209.

[14] "Linux Security Module Usage — The Linux Kernel documentation." https://www.kernel.org/doc/html/v5.0/admin-guide/LSM/index.html.

[15] Eddie BILLOIR, "drop-dac-override." Apr. 19, 2023. https://GitHub.com/LeChatP/drop-dac-override

[16] P. Samarati and S. C. de Vimercati, "Access Control: Policies, Models, and Mechanisms," in Foundations of Security Analysis and Design, Berlin, Heidelberg, 2001, pp. 137–196.

[17] "Home — TCPDUMP & LIBPCAP." https://www.tcpdump.org/

[18] 'BPF Compiler Collection (BCC)'. IO Visor Project, May 28, 2023. https://github.com/iovisor/bcc

[19] Eddie BILLOIR, "kapable-clang-sast." Apr. 17, 2023. https://GitHub.com/LeChatP/kapable-clang-sast

[20] "sudo-project/sudo.c" *sudo* Project, May 21, 2023. https://GitHub.com/sudo-project/sudo

[21] M. Abedin, S. Nessa, L. Khan, and B. Thuraisingham, "Detection and Resolution of Anomalies in Firewall Policy Rules" in Data and Applications Security XX, E. Damiani and P. Liu, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, doi: 10.1007/11805588_2.

[22] "Rust Programming Language." https://www.rust-lang.org/.

[23] "Abstract syntax tree" Wikipedia. Jan. 27, 2023. https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1135931906

[24] R. Laborde, A. Oglaza, A. S. Wazan, F. Barrère, and A. Benzekri, "A situation-driven framework for dynamic security management", Annals of Telecommunications, vol. 74, pp. 185-196, 2019.

[25] M. Krohn et al., "Make Least Privilege a Right (Not a Privilege)".

[26] M. S. Miller, "Towards a Unified Approach to Access Control and Concurrency Control".

[27] Hallyn, Serge E and Morgan, Andrew G, "Linux capabilities: making them work"