Implementing the Principle of Least Administrative Privilege on Operating Systems: Challenges and Perspectives

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

Implementing the Principle of Least Administrative Privilege on Operating Systems: Challenges and Perspectives

Eddie Billoir[1,3*], Romain Laborde[1], Ahmad Samer Wazan[2], Yves Rütschlé[3], Abdelmalek Benzekri[1]

[1] IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France.
[2]Zayed University, Academic City, Dubai, United Arab Emirates.
[3]Airbus Protect, 37 Avenue Escadrille Normandie Niemen, Blagnac, 31700, Occitanie, France.

*Corresponding author(s). E-mail(s): eddie.billoir@airbus.com;
Contributing authors: romain.laborde@irit.fr; ahmad.wazan@zu.ac.ae;
yves.rutschle@airbus.com; abdelmalek.benzekri@irit.fr;

Abstract

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. This article explores the different approaches implemented by the main operating systems (namely Linux, Windows, FreeBSD and Solaris) to control the privileges of system administrators in order to enforce the Principle of Least Privilege. We define a set of requirements to manage these privileges properly, striving to balance adherence to the Principle of Least Privilege and usability. We also present a deep analysis of each administrative privilege system based on these requirements and exhibit their benefits and limitations. This evaluation also covers the efficiency of the currently available solutions to assess the difficulty of performing administrative privileges management tasks. Following the results, the article presents the RootAsRole project, which aims to simplify Linux privilege management. We describe the new features introduced by the project and the difficulties we faced. This concrete experience allows us to highlight research challenges.

Keywords: Access Control, Principle of least privilege, Operating system, Administrative privileges, Linux, Windows, FreeBSD, Solaris

1 Introduction

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

On the one hand, POLP is essential from a security point of view to minimise the potential attack surface and reduce the damage in case of a security breach. It is the cornerstone of modern security models such as the zero-trust security strategy [2], which sets 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].

Historically, the operating systems (OS) were designed to have one administrator, but today 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. The co-administration of devices within the organisation or outsourcing to third parties is another illustration of this need. Although high-end proprietary software solutions may be able to meet these requirements, OSs are not designed to ensure POLP compliance according to the Zero-Trust Architecture and the related regulations.

In an operating system, administrative privileges can be assimilated into a set of permissions given to processes by OSs to execute critical tasks such as adding a new user to the OS. The POLP requires allocating distributed administrative privileges to different processes even when one administrator manages the device. However, as processes can have permissions and processes can start another process, the operating system should permit processes and their administrators to control and delegate these permissions through the system.

In this article, we define the 8 first requirements of an ideal administrative privileges system for an operating system and compare the compliance of multiple modern OS with the identified requirements. Additionally, we assess the efficiency of each requirement for each OS in terms of usability. We selected Windows, GNU/Linux, FreeBSD, and Solaris for our study to cover a wide range of operating systems prevalent in various computing environments. MacOS was not included in this list because the kernel privilege feature implemented in the MacOS XNU kernel is restricted to Mandatory Access Control Framework (MACF) modules. The MACF module in XNU is a kernel component that provides a framework for implementing mandatory access control in macOS, allowing developers to define custom access control policies and enforce rules on system resource interactions. MACF is a fork of security modules that came from BSD. However, the MACF privilege feature may not be compiled on the system [5]. Additionally, the features books state that the latter is not present in the system [6]. It is believed that this feature is only accessible to Apple kernel developers.

Some experimental operating systems, such as sel4, the KeyKOS family of operating systems or object-capability systems [7], provide interesting security mechanisms for administrative rights management following the object-capability security model [8]. However, our research focuses on operating systems widely deployed in the industry which is not the case of these alternative systems.

This article is an extension of our work previously presented at the CSNet 2023 conference [9], which only focused on the Linux operating system. To the best of our knowledge, this work is the first to i) propose a comprehensive list of requirements for comparing the administrative privilege mechanisms of current modern operating systems and ii) evaluate the internal mechanisms implemented by four major OS (Linux, FreeBSD, Windows and Solaris). These requirements are influenced by the previous research done by Miller *et al.* [8] that we revised, adapted and completed to address current OS designs. Furthermore, our requirements also consider the different stakeholders involved in the POLP enforcement: the system designers, the system developers and the system administrators. In addition, we present the RootAsRole project that enhances the management of the administrative privileges in Linux while adding more compliance with the POLP requirements.

The rest of this article is structured as follows. Firstly, in Section 3, we introduce the purpose of administrative privilege and present 8 requirements that can be used for comparing different operating systems. Then, in Section 4, we evaluate the compliance of each operating system with these requirements and assess their efficiency in meeting them. Section 5 presents the RootAsRole project, which is improving privileges management and tries to address usability (Efficiency) issues on Linux OS. Furthermore, this section highlights some of the difficulties the project is encountering during implementation. Finally, we conclude the article and present some ideas for future work.

2 Related Work

In 2005, Krohn *et al.* [10] presented "Asnix", an imaginary operating system following the object-capability security model [11]. However, their analysis did not explore Linux's 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.* [12] presented using 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.

Linux capabilities have primarily been used for enhancing container isolation [13]. Despite these improvements, vulnerabilities in containerisation and isolation technologies have been identified [14–16], due to Linux capabilities issues. To solve these challenges, the Linux community has developed new tools and software for more effective management of Linux capabilities [17, 18].

Managing Linux capabilities is still a challenge. In [9], we tested different approaches for automatically mapping Linux capabilities within the kernel source code to the syscalls, but they produced inconclusive results. In parallel, Md Mehedi Hasan *et al.* worked on this specific topic and presented an interesting solution to map capabilities with their respective syscalls manually [19]. They demonstrated the ability to conduct static and dynamic analyses to extract syscalls from an Executable and Linkable Format (ELF) binary and identify their required and optional capabilities using the said map. Furthermore, they committed to updating this map manually in line with kernel evolutions. With those features, their tool named 'Decap' can substitute setUID-bit privileged programs to reduced capabilities set on extended attribute binaries. This tool is an interesting solution but the mapping between capabilities and syscalls is done manually which limits the reproductibility of their approach to other kernel versions. In addition, the task is laborious and prone to errors.

3 OS administrative privileges management

3.1 What are administrative privileges?

The kernel is the main component of an operating system. It is mainly responsible for:

- *The management of processes.* Features include scheduling the execution of multiple processes or tasks and managing the creation and termination of processes.
- *The memory management.* The kernel provides a virtual memory space for each process and allocates and de-allocates memory space.
- *The management of the file system.* The kernel handles the file system access and file permissions.
- *The management of devices.* The kernel includes drivers enabling software and hardware device communication and handling interruptions.
- *The communication and synchronization between processes.* The kernel provides Inter-Process Communication features, a network stack implementation and synchronisation mechanisms like semaphore and mutex.
- *The security management.* The kernel provides security features to protect the previous items.

System calls, or syscalls, are specific entry points that enable processes to perform a range of dedicated tasks, such as reading or writing files, initiating new processes, and more[20, 21]. These calls are an essential feature of operating system kernels that facilitate efficient and secure communication between processes and the underlying system resources.

```
412     if (S_ISDIR(inode->i_mode)) {
413         /* DACs are overridable for directories */
414         if (!(mask & MAY_WRITE))
415             if (capable_wrt_inode_uidgid(idmap, inode,
416                                          CAP_DAC_READ_SEARCH))
417                 return 0;
418         if (capable_wrt_inode_uidgid(idmap, inode,
419                                      CAP_DAC_OVERRIDE))
420             return 0;
421         return -EACCES;
```

Figure 1 The Linux kernel source code to bypass file access control

Operating System administrative privileges are a set of advanced permissions that allow users to access kernel services that can impact the

entire system. These privileges are required to perform specific actions, such as binding a system TCP port with a number less than 1024, altering the expected behaviour of the OS kernel, or bypassing protection mechanisms like file system access control rules. These privileges are mandatory for administration tasks; usually, the "root" or "administrator" owns all the administrative privileges.

For example, Figure 1 is an extract of the Linux kernel source code that allows a process with the CAP_DAC_OVERRIDE administrative privilege to bypass the file-system access control mechanism for directories (lines 418-420). As a result, this enables root processes with this privilege to perform any action on any directory.

3.2 OS administrative privileges enforcement architectures

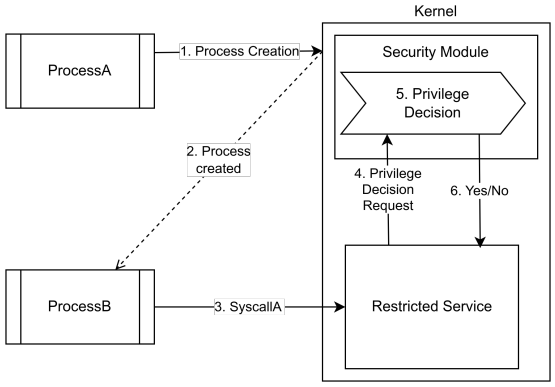

Figure 2 Permission-based enforcement architecture

An OS can implement different strategies to enforce administrative privileges: the pure permission-based strategy, the pure capability-based strategy, and a combination of permission-based and capability-based strategies. In a pure permission-based system (Figure 2), privilege decisions are taken at request time. In this case, when a user space process performs a syscall (e.g., ProcessB executes syscallA in step 3), a request will be made to a security module for each restricted section of code of the kernel to check whether or not the process can benefit from the

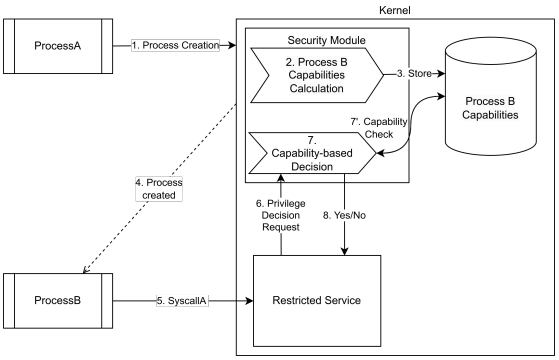

Figure 3 Capability-based enforcement architecture

specific associated functionality (Step 4). The security module will make decisions based on a security policy previously specified by the security administrators (Step 5) and return its decision to the restricted service (Step 6). The second way to manage these administrative privileges is called the capability-based approach. This approach distributes *capabilities* that grant privileges through processes (Figure 3). In this case, administrative privilege decisions are made during process creation. Each process owns a set of capabilities inherited from its parent and maintained by the kernel. A process can retrieve its list of capabilities and revoke a subset of its capabilities. When ProcessA creates ProcessB (Step 1), the security module calculates the capabilities of ProcessB based on the ProcessA's capabilities and potential external policies (Step 2). It stores them in a dedicated structure provided by the kernel (Step 3). When ProcessB executes a syscall (Step 5), a request is sent to the security module to check if the process owns the required capabilities (Step 6). The security module can then check the capabilities of the calling process from the dedicated kernel structure (Steps 7 and 7') to provide its decision.

In the first approach, the system needs to process a decision for each request, whereas the decision is made at process creation and needs to be maintained in memory in the second approach. Thus, permission-based systems require more processing, while capability-based systems need more memory to store decisions. However, both approaches are interesting, and some solutions combine decisions from multiple security modules[22].

3.3 OS administrative privileges requirements

We propose 8 requirements to define an ideal OS administrative privileges system. Dynamic privileges management requirements were adapted from [8]. They allow us to compare the different mechanisms implemented by operating systems. These requirements consider two system elements: a program (or its instantiation as a process) and a thread. A program possesses the capability to execute another program, thereby substituting the current process image with a new one. Similarly, a thread can duplicate itself to generate a new thread. Both of these activities establish a hierarchical relationship. For instance, a parent program executes a child program, while a parent process duplicates its child process. The requirements are explained below:

A. *Granularity*

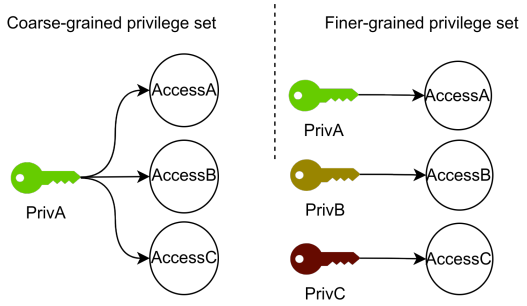

Figure 4 Privilege Granularity representation

Since POLP is about minimising the privileges, the initial focus is on Granularity, shown in Figure 4. Indeed, POLP cannot be implemented if the kernel provides only coarse-grained privileges. Therefore, one administrative privilege must refer to one action on one kernel service. The granularity requirement is not achieved if an administrative privilege covers multiple actions on objects. For instance, if a privilege permits bypassing both read and write file access controls, it should be split into two separate privileges, one for action read and one for action write. Additionally, we evaluate the precision of action verbs; for instance, "perform administration tasks" is vague and covers multiple kernel services. Permission "Change the time zone" is more precise, specifying one specific action on a particular kernel service.

B. *Uniqueness*

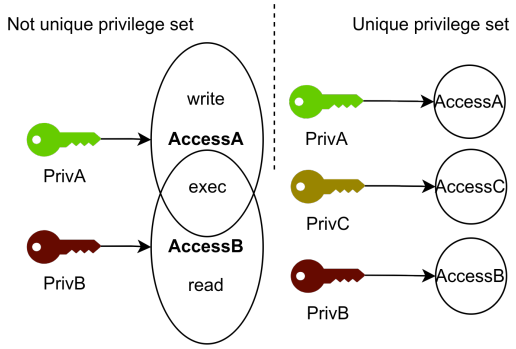

Figure 5 Privilege Uniqueness representation

Our second requirement is Uniqueness, which means the same access cannot be granted by two different privileges. For example, Figure 5 shows on the left side that both privileges A and B grant an "exec" access; this example creates a dilemma on which privilege is least privileged. Ensuring uniqueness avoids such issues. Proving Uniqueness is challenging [9], but it is easy to detect a counter-example. Checking directly in the documentation or source code can detect a non-unique privilege set.

C. *Enforcement Objects*

The operating system must efficiently grant privileged access by specifying enforcement objects. For instance, privileges can be enforced at the user level when assigning privileged users. Here, the enforcement object is the users. Adminstrative privileges can be enforced on diverse objects (e.g., user ID, group ID, file) to support POLP, ensuring a balanced approach between security and usability.

D. *Dynamic Initialisation*

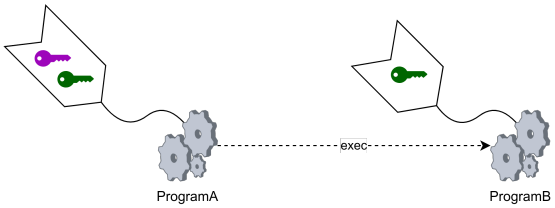

Figure 6 A program starts another program with fewer privileges

Dynamic initialisation requires the ability to grant specific privileges when new programs are executed (i.e., when the current process image is replaced with a new process image), allowing on-the-fly management of program dependencies, such as launching a background service with reduced permissions when triggered by a user-initiated action, see Figure 6.

E. *Init Authorisation Verification*

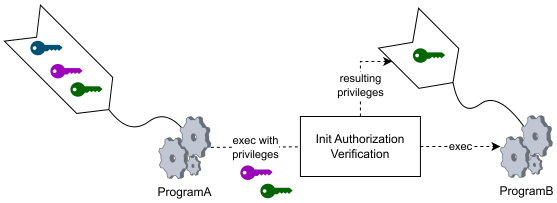

Figure 7 A program attempts to launch another program with two privileges, but the administrator grants only one.

To validate *Dynamic Initialisation*, administrators must be able to inspect and curtail privileges granted during program initialisation, such as reviewing and adjusting access rights assigned to a newly instantiated application, see Figure 7. For the POLP objective, we specify that this requirement should only reduce inherited privileges and not grant new ones.

F. *Dynamic Delegation*

Dynamic delegation means enabling threads (or processes) to delegate a privilege to a new thread or a new sub-process that allows privileges to be turned on/off at any time (see Figure 8). However, these privileges

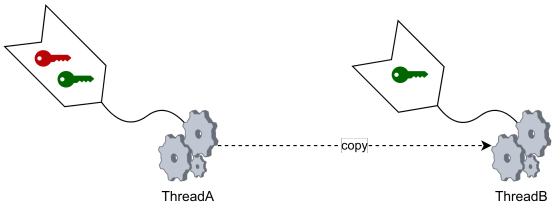

Figure 8 A thread starts another thread with fewer privileges

should not be enabled by default. Threads must explicitly activate the required privileges to specify the minimum time-of-use and thus align with the POLP objective.

G. *Self-Revocability*

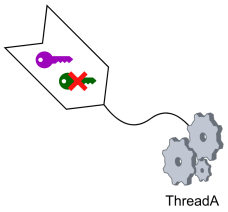

Figure 9 A thread revokes a privilege to itself

The self-revocability requirement allows a thread to permanently revoke a privilege for itself, preventing a new thread or new sub-process from obtaining these privileges, see Figure 9.

By incorporating *Self-Revocability* and *Dynamic Delegation* requirements, a thread can adhere to the POLP for operating system privileges. For instance, consider a process initially possessing three privileges. If this process spawns three sub-processes, each inheriting one privilege, and subsequently, the parent process revokes its privileges, it results in a process tree where only the child processes retain privileges. Significantly, these child processes are limited to using their respective privileges, and none of them can wield multiple privileges, as the parent process has revoked them, as presented in Figure 10.

H. *Mandatory Runtime Revocation*

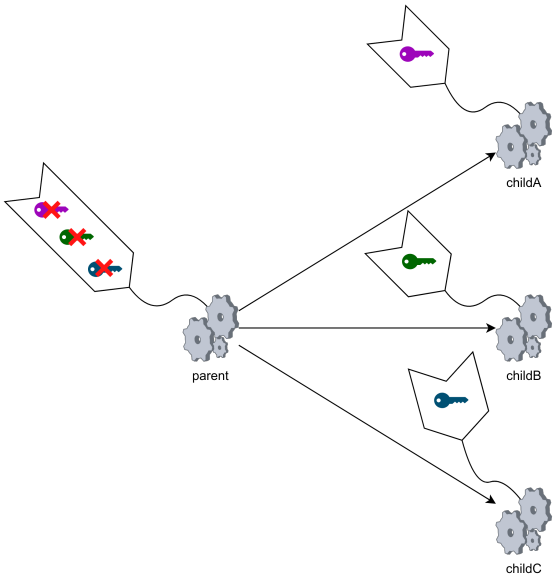

Figure 10 POLP applied by parent-revoking and Dynamic Delegation

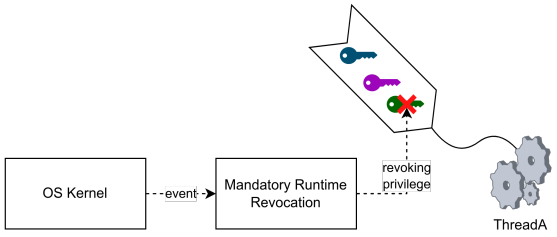

Figure 11 Revocation triggered by kernel event that revokes a privilege of a thread

This requirement allows an administrator or system to verify and revoke privileges for any thread at any time. This feature can be useful for a system that needs to dynamically restrict access to privileges and implement a dynamic separation of privileges. This requirement further elaborates on POLP and addresses specific security needs. For instance, on Linux, the use of multiple privileges can create vulnerabilities [23]. Revoking the other privileges when one of them is used can be an effective security measure.

3.3.1 Usability Considerations

While these requirements offer valuable security properties, they do not guarantee usability [24]. Security services must be user-friendly for developers, administrators, and even inexperienced users. However, previous requirements items concern developers and administrators. We will not consider that an inexperienced user can enrol in an administrator job, as we evaluate *Efficiency* and not the *Learnability*. For Efficiency, we assess the number of steps needed to complete a task [25]. These steps could include typing a command line, a mouse click, configuring a value in a file, or writing one line in the source code of a program. For this comparison, we distinguish two roles: Administrator and Developer. This distinction is based on the unique considerations for programs created by developers and the responsibilities of administrators overseeing them.

For program's Developers, we are looking at two levels of implementation: userland API and kernel-internal API. If the goal can be achieved in userland, counting the steps using syscalls is straightforward. If it requires using a kernel-internal API, we cannot determine the number of steps because it involves processes beyond our current scope, like comparing the kernel recompilation process. It's also possible that there isn't a kernel-internal API designed for our goal, necessitating a complex kernel modification from the start. For this article, our initial focus is on the implementation of program requirements, including *Dynamic Initialisation*, *Dynamic Delegation* and *Self-Revocability*. According to the previous paragraph, as we evaluate *Efficiency*, we specifically analyse the number of source code lines necessary for their fulfilment. To ensure a fair comparison, we only count lines of code in the C language. We use the *cloc* tool to count C code lines in our examples [26].

For administrators, Linux, FreeBSD, and Solaris kernels do not independently handle user interaction. Therefore, our comparison utilises a basic GNU/Linux setup with minimal user interaction tools for Linux capabilities. We consider FreeBSD and Solaris only minimal installations. This article examines the availability of comprehensive tools to configure proposed services. This assessment determines the number of actions an administrator requires to enforce the designated

Enforcement Objects. Additionally, we analyse the Efficiency in configuring *Mandatory Runtime Revocation* and *Init Authorisation Verification*, considering the number of commands involved. When an operating system does not comply with requirements, we explore its feasibility by a developer, considering the criteria outlined in the preceding paragraphs.

4 OS privileges management implementations

4.1 Linux Capabilities

Linux divides the privileges associated with the superuser into distinct units called capabilities. These capabilities enable privileges to be assigned to individual applications or tasks without the need for them to run with root, the superuser account. They initially expressed in implementing the POSIX 1003.1e draft standard [27]. However, this draft was withdrawn, allowing Linux kernel developers to modify, maintain and enhance it.

4.1.1 Linux Requirements evaluation

The Linux capabilities can grant access to kernel services or bypass 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 does not define any scope; this decision was offering flexibility for kernel designers to manage capability definitions and keeping a better performance goal [28]. Regarding our Granularity requirement, there is an important issue with Linux. Out of the *CAP_SYS_ADMIN* case, we can observe that *CAP_NET_ADMIN* and *CAP_SYS_RESOURCE* also allow many actions on the system without a capability for each of them. While these actions may be very specific, they share multiple use cases. However, we can also read that *CAP_WAKE_ALARM* is a privilege needed only to wake up the system based on a timer, which is very specific in contrast to previous ones. Based on this information, we can say that Linux has an inconsistent granularity because some privileges are general and overloaded, and some are very precise without valid documentation justification. Even the documentation explains that there are mistakes in capabilities assignation management [29].

Our previous work [9] showed that the requirement for *Uniqueness* was not fully met. This is because some privileges have the same access. For instance, *CAP_DAC_OVERRIDE* and *CAP_DAC_READ_SEARCH* privileges have overlapping access, and their lack of fine granularity and uniqueness is evident. The former privilege allows bypassing reading, writing, finding, executing, and many more actions on the file system, while the latter enables bypassing reading and finding any file on the system. Although this design makes sense in specific use-case scenarios, such as finding text occurrences on the entire system, choosing a specific privilege for this particular use-case is historical.

The Linux capability system relies on two Enforcement Objects: identity and file. Identities like user or group identifiers can be used as enforcement objects using the PAM 'pam_cap.so' module to grant privileges to user sessions. Binary files are enforcement objects using Linux extended attributes that grant them privileges. Any other criterion may be an additional software layer to these enforcement objects. For example, RootAsRole is our current software that implements a Role-based Access Control model to increase the Linux credentials manipulation usability, including the Linux Capabilities. However, the privileges are granted to the binary through extended attributes, making file-system compatibility dependent.

The kernel maintains 5 sets of capabilities in memory to manage POLP dynamically: Effective, Permitted, Inherited, Ambient, and Bounding. These sets define the state of the discretionary and mandatory privilege policy. With Effective, Permitted and Bounding sets, developers can use privileges with a time-of-use criterion and distribute them across the child threads. These three sets comply with *Dynamic Delegation* and *Self-Revocability* requirements. With Inherited, Ambient and Bounding sets, developers and administrators can determine how privileges should be inherited when executing another binary. These

three sets comply with *Dynamic Initialisation* requirement.

However, these sets do not fulfill *Init Authorisation Verification* requirement because administrators cannot refuse capabilities inheritance. Administrators can implement this feature using a Linux Security Module (LSM). LSM is a mandatory access control feature that makes decisions on access control to all resources and features in the kernel. An LSM can use the *bprm_creds_from_file* hook[1]. An LSM hook is a specific point within the Linux kernel where security modules can intercept and control system calls or other operations to enforce security policies. However, this hook is not accessible through the SELinux or AppArmor LSMs. An administrator would thus have to rely on a new LSM. There are two ways to create an LSM on Linux: one with enhanced Berkley Packet Filter feature (BPF_LSM) and one through kernel development and recompilation. However, the BPF_LSM is exclusively read-only, not allowing a process to edit capabilities. The last option is to develop and recompile the kernel with this new LSM; as a consequence, we can say that the kernel does not comply with *Init Authorisation Verification* requirement.

Also, Linux theoretically complies with *Mandatory Runtime Revocation* as LSM can intercept most events on the kernel and deny privilege requests. SELinux and AppArmor, however, do not modify the credentials structure, so they do not allow administrators to revoke capabilities granted at initialisation.

4.1.2 Linux Usability evaluation

For a Developer's program, *Dynamic Initialisation* is automatic, as the original program may be initiated directly with correct Inheritable privilege sets. Additionally, this manner completely complies with *Dynamic Delegation* rule, which privilege should not be enabled by default. However, if the automatic way is inappropriate for the program use cases, a program may need to add privilege in the Ambient set. *libcap* library is the recommended way to perform it [30]. Figure 12 presents how to raise ambient capabilities before running 'execve()'. This implementation needs 14 lines of C code without the 'execve()' call.

[1]The *bprm_creds_from_file* hook allows editing inherited capabilities when a program calls a binary

```c
int exec_with_ambient_caps(char *program,
                           char *argv[],
                           char *envp[],
                           cap_value_t caps[],
                           int n_caps) {
    // Set the ambient capabilities
    for (int i = 0; i < n_caps; i++) {
        if (cap_set_ambient(caps[i], CAP_SET) == -1) {
            return 1; // Return 1 on failure
        }
    }

    // Execute the new program
    execve(program, argv, envp);

    // If execve returns, it means there was an error
    return 2;
}
```

Figure 12 Set ambient capabilities before executing program

```c
#include <sys/capability.h>
int remove_all_eff_caps(void){
    // Get the current process's capabilities
    cap_t caps = cap_get_proc();
    if (!caps)
        return 1;
    // Iterate through all the capabilities
    for (cap = 0; cap <= CAP_LAST_CAP; cap++) {
        cap_flag_value_t flag;
        // Remove the effective capability
        ret = cap_set_flag(caps, CAP_EFFECTIVE, 1, &cap, CAP_CLEAR);
        if (ret) {
            cap_free(caps);
            return 2;
        }
    }
    // Setting Cleared effective set
    ret = cap_set_proc(caps);
    if (ret) {
        cap_free(caps);
        return 3;
    }
    cap_free(caps);
    return 0;
}
```

Figure 13 A clearing Effective Set function from current capabilities sets

Concerning the *Dynamic Delegation* requirement, it is also automatic on Linux; as a developer decides to create a thread, Linux duplicates the caller capabilities sets. However, if they are effective in the caller, sub-processes will also have theirs, which does not automatically comply with our requirements. To be fully compliant, a developer must clear the Effective set before creating a sub-thread; see Figure 13 above. As we explained in Section 3.3.1, the example is iterating through all capabilities as they need to be cleared individually with this API. This example needs 21 lines of C code.

```c
#include <sys/capability.h>
int revoke_privileges(void) {
    // Get the current process's capabilities
    cap_t caps = cap_get_proc();
    //set SETPCAP capability effective
    cap_value_t setpcap = 8;
    cap_set_flag(caps, CAP_EFFECTIVE, 1, &setpcap, CAP_SET);
    cap_set_proc(caps); // set setpcap effective
    // Iterate through all the capabilities
    for (int cap = 0; cap <= CAP_LAST_CAP; cap++) {
        // Drop the bounding capability
        cap_drop_bound(cap);
        // Clear the ambient capability
        cap_set_ambient(cap, CAP_CLEAR);
        // Clear the inheritable, permitted and effective capabilities
        cap_set_flag(caps, CAP_INHERITABLE, 1, &cap, CAP_CLEAR);
        cap_set_flag(caps, CAP_PERMITTED, 1, &cap, CAP_CLEAR);
        cap_set_flag(caps, CAP_EFFECTIVE, 1, &cap, CAP_CLEAR);
    }
    // set no new priv to the process
    prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);

    // disable and lock the securebits about capabilities granting
    prctl(PR_SET_SECUREBITS,
                SECBIT_KEEP_CAPS_LOCKED | SECBIT_NO_SETUID_FIXUP |
                SECBIT_NO_SETUID_FIXUP_LOCKED | SECBIT_NOROOT |
                SECBIT_NOROOT_LOCKED);
    // Set the new capabilities
    cap_set_proc(caps);
    return 0; // Return 0 on success
}
```

Figure 14 Proper and permanent capabilities revocation

Concerning *Self-Revocability*, the program will need CAP_SETPCAP privilege as a prerequisite to revoke its privileges. This prerequisite is for a specific security reason [31]. Furthermore, to prevent any privileges from being granted in any other way after revocation, Linux requires to turn on multiple security bits to the current process credentials (Line 24 on Figure 14). Thus, revocation is definitive. In Figure 14, we summarised the implementation for this article's clarity; a proper implementation would require memory and error management (and more header includes). The entire implementation of this code is about 58 lines of code.

```
auth     optional     pam_cap.so
```

Figure 15 Line to invoke pam_cap.so during PAM authentication in PAM configuration files

```
cap_net_raw    alice
```

Figure 16 Line to grant Alice the CAP_NET_RAW privilege in pam_cap.so capability.conf file

As we introduced users/groups in *Enforcement Objects*, to grant a capability to a user, an administrator needs to add the 'pam_cap.so' module to the authentication configuration in Figure 15, then configure a line to define which privilege to grant to Alice. So there are 3 steps to grant a privilege to a user: 1 for the module initialisation, 1 for specifying granted privileges and one to specify the username or group name.

```
setcap "cap_net_raw=ep" /bin/ping
```

Figure 17 Command to grant cap_net_raw to ping program

As we introduced programs in *Enforcement Objects*, an administrator needs to perform the 'setcap' command with the capabilities to grant them to a program (see Figure 17). Following the same calculation, there are 3 steps: one for the command syntax, one for the privileges and one for the binary path.

Now, we evaluate how system administrators can implement *Init Authorisation Verification*; however, its current implementation is not feasible, as detailed in the previous section. For administration developers, fulfilling this requirement entails creating an entire legacy LSM (Linux Security Module). Quantifying the number of steps necessary is challenging due to the substantial work involved in meeting this requirement. Furthermore, considering that an LSM operates on the kernel side, the feasibility is deemed to be challenging.

4.1.3 Linux conclusion

In summary, Linux lacks a well-defined set of privileges, posing challenges for the overall system. Nevertheless, the operating system incorporates noteworthy security mechanisms that dynamically allow programs to manage their privileges under administrator authorisation. However, these security mechanisms could be enhanced to permit more control and usability for administrators.

4.2 Solaris Capabilities

Solaris presents an interesting approach for distributing root privileges to Solaris users. Their approach has been stable since it was defined in 2003. The approach adopted the role-based access control model for defining roles that Solaris users can assume. RBAC implementation for Solaris differs from the NIST standard proposed by David F. Ferraiolo and al [32]. Indeed, RBAC Solaris roles

assign a set of profiles. A profile on Solaris is a set of permissions. So, roles assign a set of subsets of permissions instead of just a set of permissions. In this OS, the root account is a role that users can assume. More precisely, roles are special user accounts that cannot log in; thus, each role has an associated password that can be shared by a group of actors that share the same role.

4.2.1 Solaris Requirements evaluation

Privileges	Objects criteria
file privileges, proc_exec	Filename, Wildcarded File
net_privaddr	Network port, Range of ports
proc_setid	Username, Uid, Uid range

Table 1 Available privileges extension in Solaris 11.4

In Solaris 11.4, the latest version, there are 90 privileges. They are finer-grained than Linux (e.g., discretionary file access control privileges that are adequately separated) but still regroup a list of actions that can be imprecise. For example, there is the sys_config privilege that has an imprecise description with "various system configuration tasks" term that does not precisely describe which action an administrator can perform on the system. However, administrators can refine some resources for some privileges with more precision. For example, administrators can limit the use of net_privaddr (which allows any program to bind to a port less than 1024) to port 80 and TCP protocols. This mechanism applies to every privilege associated with specific object types such as users, network ports, and files as described in Table 1. Thus, it allows precise privilege usage contexts. However, this feature does not solve the granularity issue of other unprecise privileges. However, with better granularity than Linux, Solaris still has an incorrect *Granularity*. Given that *Granularity* is incorrect and it is related to *Uniqueness*, we cannot know if *Uniqueness* is reached.

Solaris presents a different approach to managing privileges. Instead of storing privileges in the extended attributes of the executable, Solaris provides administrators with a set of central databases (user_attr, prof_attr, exec_attr and auth_attr) for enforcing privileges based on a user or roles only. Indeed, it is impossible to grant privileges to a group directly; the only way to do it is through a role.

From the kernel point of view, Solaris defines 4 privilege sets for each process: Effective set, Permitted set, Inheritable set, and Limit set. Effective and Permitted sets are handled in the same manner as Linux. However, Inheritable and Limit sets are calculated differently during the 'exec()' call. This implementation complies in a very similar way to Linux, thus leading to an equivalent compliance in dynamic privilege management.

On Solaris, Trusted Extension is the mechanism to manage mandatory multi-level security policy and general mandatory access controls on the kernel. However, it does not dynamically manage *Mandatory Runtime Revocation* because their policy is fixed through "zones" that cannot be modified at runtime [33].

4.2.2 Solaris Usability evaluation

```c
#include <priv.h>

int set_inheritable_network_privileges() {
    priv_set_t *privileges;

    // Allocate a new privilege set
    privileges = priv_allocset();
    if (privileges == NULL) {
        return -1; // Allocation failed
    }

    // Empty the privilege set
    priv_emptyset(privileges);

    // Add the network privilege to the set
    if (priv_addset(privileges, "net_privaddr") == -1) {
        priv_freeset(privileges);
        return -1; // Failed to add privilege
    }

    // Set the process inheritable privileges to the network privilege set
    if (setppriv(PRIV_SET, PRIV_INHERITABLE, privileges) == -1) {
        priv_freeset(privileges);
        return -1; // Failed to set privileges
    }

    // Free the privilege set
    priv_freeset(privileges);

    return 0; // Success
}
```

Figure 18 set inheritable privileges on current process

For a developer's program, as Linux does, *Dynamic Initialisation* occurs automatically when the original program is initiated directly with the correct inheritable privilege sets. Unlike Linux, it is automatically set to the Effective set, enabling privileges by default, which does not fully comply with our requirements. If the automatic method is unsuitable for the program's use case, the program may need to adjust privileges in the

inheritable set, managed through the 'setppriv()' syscall. Figure 18 illustrates how to adjust inheritable privileges. In Solaris, unlike Linux, a process has a default set of privileges, and implementing this example would also remove 'basic' privileges, which are required to perform many 'basic' actions that are not specific to administrative tasks (e.g., create a sub-process). The code for this example consists of 19 lines in C.

As for Linux, *Dynamic Delegation* is automatic and does not fully comply with our requirements. To achieve full compliance, developers must clear their Effective set before creating a sub-thread. This requirement can be done similarly to the approach shown in Figure 18. The subtle difference is to remove lines 16-19 and set PRIV_EFFECTIVE instead of PRIV_INHERITABLE in the 'setpriv()' syscall. Without these lines, the modified example is 15 lines of C code.

```c
#include <priv.h>
#include <unistd.h>

int revoke_privileges(void) {
    priv_set_t *empty_set;

    // Allocate an empty privilege set
    empty_set = priv_allocset();
    if (empty_set == NULL) {
        return -1; // Allocation failed
    }

    // Empty the privilege set
    priv_emptyset(empty_set);

    // Set the process privilege sets to the empty set
    if (setppriv(PRIV_SET, PRIV_PERMITTED, empty_set) == -1 ||
        setppriv(PRIV_SET, PRIV_INHERITABLE, empty_set) == -1 ||
        setppriv(PRIV_SET, PRIV_EFFECTIVE, empty_set) == -1 ||
        setppriv(PRIV_SET, PRIV_LIMIT, empty_set) == -1) {
        priv_freeset(empty_set);
        return -1; // Failed to set privileges
    }

    // Free the privilege set
    priv_freeset(empty_set);

    return 0; // Success
}
```

Figure 19 Revoke privileges on current process

Concerning *Self-Revocability*, Figure 19 removes every set's privileges. This example requires 19 lines of code, similar to previous requirements.

```
usermod -K defaultpriv=basic,net_privaddr alice
```

Figure 20 Command to grant net_privaddr to user alice

As we introduced the user in *Enforcement Objects*, to grant a capability to a user, an administrator needs to perform the 'usermod' command by specifying parameters as shown in Figure 20. So there are 3 steps: 1 for the command syntax, 1 for the user name, and lastly, the privileges to grant.

```
ebilloir@solaris:~$ sudo profiles -p "tcpdump right profile"
Mot de passe :
profiles:tcpdump right profile> set desc="tcpdump right profile"
profiles:tcpdump right profile> add cmd=/usr/sbin/tcpdump
profiles:tcpdump right profile:tcpdump> add privs=net_rawaccess
profiles:tcpdump right profile:tcpdump> end
profiles:tcpdump right profile> commit
profiles:tcpdump right profile> exit
ebilloir@solaris:~$ sudo roleadd -c "Tcpdump role" -K profiles="tcpdump right profile" r_tcpdump
ebilloir@solaris:~$ sudo usermod -R +r_tcpdump ebilloir
UX: usermod: ebilloir is currently logged in, some changes may not take effect until next login.
ebilloir@solaris:~$ su r_tcpdump
Password:
r_tcpdump@solaris:~$ tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on net0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C
0 packets captured
0 packets received by filter
0 packets dropped by kernel
r_tcpdump@solaris:~$ exit
exit
ebilloir@solaris:~$ tcpdump
tcpdump: net0: You don't have permission to capture on that device
((cannot open device) /dev/bpf: Permission denied)
```

Figure 21 Role r_tcpdump creation, assigned to ebilloir, to perform tcpdump with net_rawaccess

As we introduced Roles in *Enforcement Objects*, Figure 21 shows how to create a 'right profile' that designates tcpdump binary with net_rawaccess privilege. This right profile is assigned to a new role named 'r_tcpdump', then being assigned to ebilloir. This example allows ebilloir to switch to this role and use tcpdump. This screen does not show that it requires setting the r_tcpdump password. Following each user-input screen line, it needs :

Line 1. 2 steps
Line 2. 1 step
Line 3. 2 steps
Line 4. 2 steps
Line 5. 2 steps
Line 6. 1 step
Line 7. 1 step
Line 8. 1 step
Line 9. 4 steps
Line 10. 3 steps

The following lines are just for testing purposes. In total, 19 steps are required to set up a Role on Solaris. Adding the missing 'sudo passwd r_tcpdump' command is 2 additional steps, and changing the password is 2 steps, as you need to provide and confirm the password. In total, there are 23 steps.

Finally, we did not find an API for *Program Enforcement Object*, *Init Authorisation Verification* and *Mandatory Runtime Revocation*, as the kernel source code is not available.

4.2.3 Solaris conclusion

Solaris privileges are more accessible than Linux because Enforcement Objects are centralised and mandatory. They are more usable than Linux because of their more precise Granularity but less understandable than Linux because of their imprecise documentation. These implementation mechanisms make our administrator usability better than Linux but still need significant effort to make them easy to use.

4.3 Microsoft Windows Privilege Constants

For Microsoft Windows products, we are evaluating Windows 10/11 Pro. Windows 10/11 Home editions don't have the necessary tools for our comparison; we explain that below in Section 4.3.2. Also, we don't consider Windows Server products as the main difference between them is that they are local-only solutions for Windows 10/11 and distributed solutions with many additional distributed access control models for Windows Server. As these additional models are accessible only through OS network directory services, they are not considered in this comparison. Microsoft Windows utilises "privilege constants" [34]. These privileges serve as criteria for permissions in different access control models, as presented by Microsoft through tools like "Local Security Policy" for individual systems or "Local Group Policy Editor" for the distributed version.

4.3.1 Windows Requirement evaluation

While Windows privileges exhibit a relatively fine-grained set of privileges, we did not notice any evident granularity issue on their documentation. Windows present 36 privileges linked to a specific user-level use case (e.g., "Back up files and directories"). Windows being closed-source, we cannot assess it. These privileges rely on finer-grain privileges that are unavailable from the user's point of view. Thus, we can also not evaluate the *Uniqueness* requirement at this stage. However, Granularity is sufficient for general Windows system usage. Indeed, suppose user Alice wants to back up her system. In that case, Alice needs the SeBackupPrivilege without worrying about another privilege because the system will be in a particular state that allows its entire backup.

On Windows, the Identity is the unique enforcement type for privilege management. A Windows Identity can be a User, Group, Computer system or Service. When a privilege is bestowed on an identity, it possesses every privilege for every initiated session. This exclusive feature poses a challenge in the context of the POLP, as administrators are compelled to preemptively grant users a broad range of privileges or implement an interface script to change the user identity to perform tasks temporarily. Another challenge is managing the authentication of every specific identity for every case.

In conclusion, while privileged identity is an interesting solution for specialised cases, making it a unique and general solution is laborious for administrators who want to respect POLP and prevent privilege escalation. This approach implies that unless the administrator's team manages a heavy business privilege access management process, most will adhere to the standard administrator configuration and not manage these privileges. So, this unique object enforcement negatively impacts the administrator's usability.

Regarding Dynamic privilege management, the Windows API allows programs to spawn another subprogram with duplicated privileges, which the parent can further restrict before spawning the child program. This complies with *Dynamic Initialisation* and *Dynamic Delegation* requirements. Also, programs can reduce privileges with the 'AdjustTokenPrivileges()' function, which complies with the *Self-Revocability* requirement.

While Linux users can grant or restrict privileges to sub-programs, Windows links privileges to the user identity and thus offers no mechanism for the user to manage privileges of their programs. In other terms, it is impossible for an administrator to revoke a privilege at any time during a process or to reduce inherited privileges to a new subprogram. So, Windows does not comply with *Mandatory Runtime Revocation* and *Init*

Authorisation Verification requirements. Indeed, Windows only allows Identity type enforcement as the only way to manage "Privileges Constants".

4.3.2 Windows Usability evaluation

In Windows, achieving *Dynamic Initialisation* and *Dynamic Delegation* is possible through the use of the 'CreateRestrictedToken()' function, while *Dynamic Revocation* can be accomplished using 'AdjustTokenPrivileges()'. John Viega and Matt Messier present many examples in their "Secure Programming Cookbook for C and C++" book section 1.2[35]. If we implement their pseudo-code with basic error handling (printing the error and returning in case of an error), the C code required for *Dynamic Initialisation* and *Dynamic Delegation* amounts to 56 lines. Moreover, *Dynamic Revocation* necessitates approximately 34 lines.

```c
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "advapi32.lib")

BOOL SetPrivilege( HANDLE hToken,
        LPCTSTR lpszPrivilege,
        BOOL bEnablePrivilege) {
    TOKEN_PRIVILEGES tp;
    LUID luid;
    LookupPrivilegeValue(NULL,lpszPrivilege, &luid);
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid = luid;
    if (bEnablePrivilege)
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    else
        tp.Privileges[0].Attributes = 0;
    // Enable the privilege or disable all privileges.
    AdjustTokenPrivileges(
            hToken,
            FALSE,
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            (PTOKEN_PRIVILEGES) NULL,
            (PDWORD) NULL)

    return TRUE;
}
```

Figure 22 pseudo-code Function to turn a privilege on or off

Additionally, for a *Dynamic Delegation*, like other OS by default, a privilege can be enabled and disabled. Turning a privilege on and off could be made by using examples provided by Microsoft, that are available on their website[36]; a simplified pseudo-code is shown in Figure 22.

In Windows, there are multiple ways to enforce privileges. In a Microsoft environment with Windows Server, administrators can utilise Global

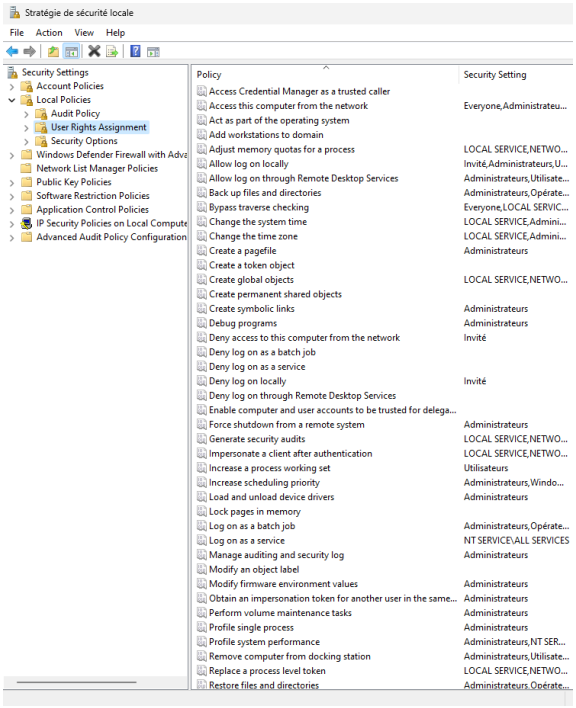

Figure 23 Local Security Policy window

Policy Objects (GPOs) to enforce these objects across the entire Active Directory, the Microsoft's centralised directory service for managing multiple systems.

For Windows 10/11 Pro, administrators can easily access security policies by pressing 'Windows + R', typing 'secpol.msc', and pressing enter. However, this method may not be available for specific Windows Home editions, which require additional steps (beyond the scope of this explanation).

Now, let us consider the process of granting the "SeDebugPrivilege" to the local user Alice. To accomplish this, an administrator needs to follow these steps:

1. Right-click on the "Debug programs" privilege
2. Left-click on "Property"
3. Left-click on "Add User or Group"
4. Left-click on "Advanced"
5. Left-click on "Find Now"
6. Double-click on Alice's username
7. Left-click on "OK"
8. Left-click on "Apply"
9. Finally, left-click on "OK"

This entire process involves 12 steps. However, it requires repeating the 9 last steps for each privilege needed.

4.3.3 Windows conclusion

Finally, Windows offers minimal management of a well-defined set of privileges. While developers have many features to manage their privileges, administrators need more control and tools. In other words, if a malicious program owns a privilege, it can do whatever it wants, and administrators cannot take effective action on the program unless they kill it.

4.4 FreeBSD Privilege

In 2006, FreeBSD introduced a privilege feature as a fine-grained access control criterion. Since its implementation, around 235 privileges have been added. These privileges are typically labelled as "PRIV_COMPONENT_ACTION" or describe an action related to a feature or component. Unlike previously evaluated operating systems, FreeBSD does not store privileges in the process credentials structure. Instead, each privilege is requested and evaluated by security modules that hook into the 'priv_check' privilege request. While mechanisms are in place in the kernel to implement these privilege checks, they are not, in fact, currently used. It is up to the system administrator to define a policy using a security module. However, their website has a remark on the capability-based model features and decided not to risk editing the current implementation to this new model[37]. As explained in Section 3.2, one model is not better than the other. Furthermore, in 2012, FreeBSD implemented Capsicum, an Object-capability security model that supports fine-grained user-sandboxing features[38]. It is not intended for administrative task management, so it is not included in our paper comparison.

4.4.1 FreeBSD Requirements evaluation

In terms of Granularity, FreeBSD aims to be the most fine-grained possible. Privilege requests are minimised in their source code, often occurring within a single function or specific condition. This design allows kernel developers to define a privilege precisely by examining the source code. However, for Uniqueness, as explained in Requirement B., it is difficult to check, and we did not notice any uniqueness issues in the source code. So, we believe it may be reached.

FreeBSD security modules determine whether to grant or deny privileges directly from every kernel information. The outcome depends on the specific implementation of the security module. In FreeBSD, the SEBSD module[39], inspired by SELinux on Linux, aims to follow a similar design. Despite SEBSD being inactive for over a decade, its properties from FreeBSD implementation are noteworthy. It exclusively manages privileges through SEBSD configuration, serving as the sole mechanism for enforcing security policies on the system. As a result, there is no specific *Enforcement Object* on the FreeBSD system. Indeed, SEBSD could arbitrarily decide to give privileged access because of a whole security context, while another security module can choose another criterion.

As processes do not have an API to do it, they cannot initiate, delegate or revoke them, thus setting Dynamic initialisation, *Dynamic delegation*, *Self revocability*, *Init Authorisation Verification* impossible. However, *Mandatory Runtime Revocation* is fully compliant as the decision is made at each request and configurable through security modules.

4.4.2 FreeBSD Usability evaluation

FreeBSD relies on its security modules for configuration but lacks a universal method for managing privileges in common scenarios. Consequently, as developers or administrators, it is challenging to determine the steps needed to meet our requirements. FreeBSD does offer kernel APIs for security modules to implement generic *Enforcement Objects*, adhere to *Init Authorisation Verification* and *Mandatory Runtime Revocation* requirements. However, achieving *Dynamic Initialisation*, *Dynamic Delegation*, or *Self-Revocability* is not directly available using an API. Indeed, creating such API also requires managing third-party programs to utilise this API, which is not included by default in the system.

Table 2 Comparison between OS of multiple privilege management requirements

	Windows 11H2	Linux 6.7	FreeBSD 14	Solaris 11.4
Granularity	yes (use-case based)	no	yes (kernel-based)	no
Uniqueness	possible	no	possible	unknown
Enforcement Objects	User/Group/ Service	User/Group, File	None[1]	User, Role
Dynamic Initialisation	yes	yes	no	yes
Init Authorisation Verification	no	no	no	no
Dynamic Delegation	yes	yes	no	yes
Self-Revocability	yes	yes[2]	no	yes
Mandatory Runtime Revocation	no	partial	yes	no

[1] Without TrustedBSD, no objects are associated with these privileges
[2] It can completely turn off all granting mechanisms

Table 3 Usability (Efficiency) comparison between OS for requirements

	Windows 11H2	Linux 6.7	FreeBSD 14	Solaris 11.4
User/Group Enforcement Object	12 Steps[1,2]	3 Steps	kernel API	3 Steps
Program Enforcement Object	N/A	3 Steps	kernel API	N/A
Role Enforcement Object	N/A	kernel API	kernel API	23 Steps
Dynamic Initialisation	56 C lines	14 C lines	no API	19 C lines
Init Authorisation Verification	no API	kernel API	kernel API	no API
Dynamic Delegation	56 C lines	21 C lines	no API	18 C lines
Self-Revocability	34 C lines	58 C lines	no API	18 C lines
Mandatory Runtime Revocation	no API	kernel API	kernel API	no API

1 This includes Service Enforcement Object
2 For each privilege

4.4.3 FreeBSD conclusion

FreeBSD's approach is interesting; it allows a well-defined and scalable privilege set without performance issues. However, it misses a by-default ability for programs to manage these privileges and the administrator's authorisation. Indeed, without these features, privileges are only statically managed for specific use cases, making the operating system security exclusive for security system designers, which is not forcibly the purpose of this operating system.

4.5 Final analysis

Finally, every OS has at least one interesting approach to managing administrative privileges. We summarize the results of our requirements evaluation in Table 2. On Windows, the granularity approach is based on use-case, while on FreeBSD, it is based on kernel needs; this shows two complementary approaches. Indeed, we could imagine an OS that develops a set of privileges at the kernel level and one at the use-case level. This design could permit kernel developers to respect POLP at their level and administrators to have a more straightforward set of privileges that can be tailored with specific kernel privileges. Additionally, the administrator should easily configure use-case-level privileges by selecting the kernel-level privileges without having to modify the kernel. This would allow one to customise the system's use cases to their usage. Uniqueness is challenging to prove; however, we found Uniqueness issues in Linux documentation and source code. Other operating systems may comply with Uniqueness, but we still need to find an algorithm to prove it.

We have observed that capability-based systems face challenges in managing dynamic privilege assignments. This is because not all requirements for dynamic privilege management are met, making it challenging for administrators to manage the system effectively. FreeBSD is the unique permission-based system tested. It is unfortunate that dynamic privilege management was not implemented on it, as the design of a permission-based system does not exclude the implementation of every requirement.

Efficiency Usability results are presented in Table 3. Linux has an extensible design that allows it to meet almost any requirement, making it advantageous compared to other operating systems. Otherwise, FreeBSD was not designed to interact with users, even with minimal tools installed. Additionally, its system requires recompiling for many requirements, a process that was not evaluated. However, like Linux, FreeBSD has an extensible design but requires a lot of kernel-level development to meet the requirements. Finally, by default, Windows requires its graphical interface exclusively to perform administrator tasks. However, graphical tasks can easily multiply and may require scripting to be accomplished effectively. Those subtlety were not evaluated in these tests.

Considering these conclusions, we cannot say that one operating system has a more complete implementation of administrative privileges than another, as their focuses differ.

5 The RootAsRole project: Improving Linux Capabilities management

In this section, we present 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.

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 [40]. *sudo* includes many other security features, but we will not elaborate on them. However, *sudo* does not handle Linux Capabilities. Since

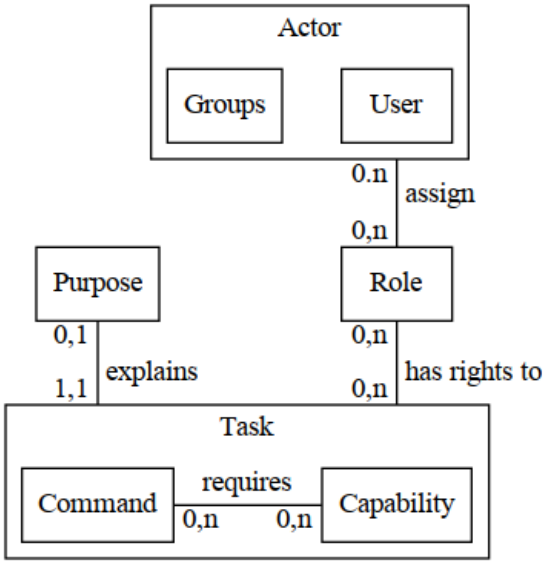

Figure 24 Design of role configuration model

no sophisticated and easy-to-use mechanisms were available to manage capabilities, we developed RootAsRole [41, 42] 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 chose to implement the Role-Based Access Control model (RBAC) that consists of granting (and restricting) access permissions to roles, and then these roles are assigned to users [43]. The RBAC model allows grouping administrative privileges and tasks by roles. We chose this model because it impels the implementation of POLP since tasks and duties must be explicitly analysed to identify roles.

To implement least privilege more precisely with RootAsRole, we took many initiatives to resolve design, usability and reliability issues for users and administrators. To illustrate these conceptual issues, we introduce a tiny web business example. Let us consider user Alice, who manages 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 configuration files, and many more web application administration tasks. 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 POLP 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 ...
```

Figure 25 A Sample RootAsRole policy for our webserver example

5.1 Administrative issues

5.1.1 Making Linux capabilities and POSIX DAC policies consistent

We developed a language that allows the administrators to specify which users can use which commands 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 24. In this model, actors, such as users or groups, are assigned roles. Permissions are assigned to roles, which are sets of commands, and grant 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 25 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 variables 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/group change for a task. The tool also manages an environment variable policy that applies a whitelist, a filter list and a define/replace list. Other variables are removed from the created session. According to various vulnerabilities found on shells, the filter list is a simple character filter, e.g., if the variable contains '%' or '/' characters, then the variable is filtered. This policy is similar to the *sudo* tool. We noticed that *sudo* enables an administrator to choose different algorithms for managing the environment variable. However, most of them are not recommended for use because of their potential vulnerabilities, e.g., CVE-2014-9680, CVE-2014-10070, and CVE-2014-0106, to name but a few. This explains why we chose to implement only the default one that is currently considered safe.

5.1.2 Capabilities are unknown by administrators

We noticed that the documentation for Linux capabilities is understandable for expert Linux developers, as they are easily contextualised with the source code. On the other hand, administrators may struggle with it. Indeed, the Linux usage manual describes some capabilities at the developer's level, but it does not explain their precise administrative use cases nor their exact purpose for each system call. For example,

```
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
```

Figure 26 Output of capable command for tcpdump

the CAP_SYS_PACCT capability has only "Use acct(2)." for description. Therefore, it was challenging 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 [41]. For example, Alice can use *capable* to determine which capability is needed for the *tcpdump* tool. Here *tcpdump* requires network capabilities, which our tool identifies and displays in Figure 26.

Capable uses extended Berkley Packet Filter (eBPF) technology to hook into the capability verification method of the kernel to collect what capabilities are requested for all processes. It then sets up an unprivileged namespace for the analysed 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.

```
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]
```

Figure 27 *bcc* Kernel stack trace describing the problematic capacity demand when using the capable tool in kernel version v4.x

```
657     /* check against resource limits */
658     if ((locked ⩽ lock_limit) || capable(CAP_IPC_LOCK))
659         error = apply_vma_lock_flags(start, len, flags);
```

Figure 28 Condition assumption example in mlock Linux feature.

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 issue was due 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 27 using the *bcc* tool [44]. Fortunately, the above problem has since been fixed in the Linux kernel. Nevertheless, this problem 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 utilises 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. For example, Figure 28 is a part of Linux kernel code for memory locking kernel feature. This feature is useful mainly for technical and performance reasons. Line 658 shows a condition where it firstly evaluates the normal system usage; then it evaluates if the thread has the privilege to bypass the IPC_LOCK limit. This is a generalized practice observed on the kernel. 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 lets us know when a capability is required because `capable()` is called when all other access control systems deny access.

During our analysis of kernel v6.3, our plugin detected 8 occurrences of noncompliance with the practice. The plugin source code can be found in a GitHub repository [45]. 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).

5.2 User usability issue

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

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

Figure 29 Substituting role (sr) command to use a role for executing tcpdump, in RootAsRole versions 1 and 2

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

Figure 30 New version of RootAsRole: Alice doesn't need to specify the role to activate

In the previous version, we required users to explicitly express the role they want to activate to execute a given command (see Figure 29). If users are assigned to multiple 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 30. However, this raises new questions, for instance, when a user is associated with multiple roles, allowing them to execute the same command but with different capabilities. There are various 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 [46] 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 granting no dangerous capability[23] is less privileged than one granting at least one of them
   (f) A role without a setuid is less privileged than one with a setuid
   (g) if no root is disabled, a role without 'root' setuid is less privileged than a role with 'root' setuid
   (h) A role without setgid is less privileged than one with setgid
   (i) A role with a single setgid is less privileged than one with multiple setgid
   (j) if no root is disabled, a role with multiple setgid is less privileged than one with set root gid
   (k) if no root is disabled, a role with root setgid is less privileged than one with multiple set gid, mainly using root group

If this algorithm does not 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 points (d) and (e), the choice of least privilege is somewhat arbitrary. In our previous research, we tried to find a partial order between Linux capabilities but could not find it[9].

6 Discussion and Future Work

We believe a formal model for administrative privileges is required. Although formalising access control for the file system was deeply studied by the research community, administrative privileges in OS have not received such attention. Jack B. Dennis and Earl C. Van Horn originally developed the concept of privilege with broader applicability beyond access control [47]. We think their definition aligns with Lampson's Identity Access Control model [48]. Jack B. Dennis and Earl C. Van Horn presented how to manage system process capabilities. Lampson's model presents a general access control matrix, a Subject × Object × Action (SOA) access control matrix. To define

an OS privilege precisely, it should be mappable onto an SOA matrix where system processes are referred to as subjects, the OS kernel features and components are referred to as objects, and explicit action verbs are referred to as actions. For example, a text editor (subject) can bypass (action) the discretionary file access control (object). This is part of our *Granularity* requirement.

Furthermore, we think OS privileges can be seen as service access control rather than a regular access control model [49]. Regular access control allows or denies an action when requested, while a service access control may not deny the request but offer a different behaviour considering these privileges. For example, on Linux, if a CAP_SYS_RESOURCE privileged process asks the operating system to create a sub-process, any resource limitations would be ignored, while an unprivileged process would need to deal with them. Also, to strengthen our comparison with service access control, we can view the operating system as a service provider that proposes different behaviours and sets security policies for allowed privileges. For our future work, we are considering formalising the notion of administrative privilege to present a management model and process that answers the initial problem: allowing untrusted co-administrated operating systems.

7 Conclusion

POLP is a well-known and established security principle. We studied different approaches to implementing POLP on diversified operating systems and demonstrated that every OS has at least one interesting approach to managing administrative privileges. Windows approach has a very simple and coherent administrative privilege set. Linux implements security mechanisms for 'by default' software usage and its extensible features through its security modules. FreeBSD implements better administrative privilege definition based on Permission-based rather than Capability-based access control. Solaris refined the Linux Capabilities to add more administrative control with RBAC, but the OS is less extensible than Linux. Considering these conclusions, we cannot say that one operating system has a more complete implementation of administrative privileges than another, as their focuses differ.

Our RootAsRole project is designed to improve the efficiency of administrative privileges management and usage on Linux. It also addresses other POLP requirements by introducing a Role Enforcement Object in Linux and a novel interactive solution for "Init Authorisation Verification" control tailored for administrators. However, we have identified certain limitations, particularly related to the user experience for both administrators and end users. Striking the right balance between usability and POLP proves to be challenging, given the coarse-grained and non-unique set of privileges involved. This conclusion aligns with the findings from our comparison of operating systems.

Conflicts of Interests

The authors have no relevant financial or non-financial interests to disclose.

References

[1] Saltzer, J., & Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, *63*(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[2] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020, August). *Zero Trust Architecture* (tech. rep.). National Institute of Standards and Technology. https://doi.org/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). (2016, April).

[4] Regulation (EU) 2021/821 of the European Parliament and of the Council of 20 May 2021 setting up a Union regime for the control of exports, brokering, technical assistance, transit and transfer of dual-use items (recast). (2022, January).

[5] Xnu/bsd/kern/kern_priv.c at 1031c584a5e37aff177559b9f69dbd3c8c3fd30a · apple-oss-distributions/xnu. (n.d.).

[6] Levin, J. (2018). **OS internals. Volume 3: Security & insecurity / by Jonathan Levin* (2nd edition). Technologeeks.com.

[7] Object-capability systems — ERights.org Wiki. (n.d.).

[8] Miller, M. S., Yee, K.-P., & Shapiro, J. (n.d.). Capability Myths Demolished, 15.

[9] Billoir, E., Laborde, R., Wazan, A. S., Rütschlé, Y., & Benzekri, A. (2023). Implementing the Principle of Least Privilege Using Linux Capabilities: Challenges and Perspectives. *2023 7th Cyber Security in Networking Conference (CSNet)*, 130–136. https://doi.org/10.1109/CSNet59123.2023.10339753

[10] Krohn, M. N., Efstathopoulos, P., Frey, C., Kaashoek, M. F., Kohler, E., Mazières, D., Morris, R. T., Osborne, M., Vandebogart, S., & Ziegler, D. (2005). Make least privilege a right (not a privilege). *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems, June 12-15, 2005, Santa Fe, New Mexico, USA*.

[11] Miller, M. S. (2006). *Robust composition: Towards a unified approach to access control and concurrency control* [Doctoral dissertation, Johns Hopkins University].

[12] Hallyn, S. E., & Morgan, A. G. (n.d.). Linux Capabilities: Making them work.

[13] Sun, Y., Safford, D. R., Zohar, M., Pendarakis, D. E., Gu, Z., & Jaeger, T. (2018). Security namespace: Making linux security frameworks available to containers. *USENIX Security Symposium*.

[14] NVD - cve-2016-0728. (n.d.).

[15] NVD - CVE-2016-8867. (n.d.).

[16] NVD - CVE-2022-27649. (n.d.).

[17] Kang, H., Kim, J., & Shin, S. (2021). MiniCon: Automatic Enforcement of a Minimal Capability Set for Security-Enhanced Containers. *2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, 1–5. https://doi.org/10.1109/IEMTRONICS52119.2021.9422529

[18] Production-Grade Container Orchestration. (n.d.).

[19] Hasan, M. M., Ghavamnia, S., & Polychronakis, M. (2022). Decap: Deprivileging Programs by Reducing Their Capabilities. *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 395–408. https://doi.org/10.1145/3545948.3545978

[20] McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. M. (2015). *The design and implementation of the FreeBSD operating system* (Second edition). Addison Wesley.

[21] Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux kernel: From I/O ports to process management* (3. ed., covers version 2.6). O'Reilly.

[22] Wright, C., Cowan, C., Morris, J., Smalley, S., & Kroah-Hartman, G. (2002). Linux security module framework. *Ottawa Linux Symposium*, *8032*, 6–16.

[23] Linux Capabilities. (n.d.).

[24] Frøkjær, E., Hertzum, M., & Hornbæk, K. (2000). Measuring usability: Are effectiveness, efficiency, and satisfaction really correlated? *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 345–352. https://doi.org/10.1145/332040.332455

[25] Rogers, Y., Sharp, H., & Preece, J. (2023, March). *Interaction Design: Beyond Human-Computer Interaction* (6th edition). John Wiley and Sons.

[26] AlDanial. (2023, December). AlDanial/cloc.

[27] Security Working Group, s. b. t. P. A. S. C. o. t. I. C. S. (1997, October). Draft Standard for Information Technology— Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API)— Amendment #: Protection, Audit and Control Interfaces [C Language].

[28] CAP_SYS_ADMIN: The new root [LWN.net]. (n.d.).

[29] Capabilities. (n.d.).

[30] Capget(2) - Linux manual page. (n.d.).

[31] Fully Capable - The Ancient Sendmail Capabilities Issue. (n.d.).

[32] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, *4*(3), 224–274. https://doi.org/10.1145/501978.501980

[33] Comparing SELinux with Solaris Trusted Extensions. (n.d.).

[34] alvinashcraft. (2022, April). Privilege Constants (Winnt.h) - Win32 apps.

[35] Viega, J., & Messier, M. (2003). *Secure programming cookbook for C and C++* (1st ed). O'Reilly.
OCLC: ocm52861976.

[36] alvinashcraft. (2023, April). Enabling and Disabling Privileges in C++ - Win32 apps.

[37] TrustedBSD - TrustedBSD POSIX.1e Privileges. (n.d.).

[38] Watson, R. N. M., Anderson, J., Laurie, B., & Kennaway, K. (2010). Capsicum: Practical capabilities for UNIX. *Proceedings of the 19th USENIX Security Symposium*.

[39] TrustedBSD - SEBSD. (n.d.).

[40] Sudo-project/sudo. (2023, May).

[41] Wazan, A. S., Chadwick, D. W., Venant, R., Billoir, E., Laborde, R., Ahmad, L., & Kaiiali, M. (2022). RootAsRole: A security module to manage the administrative privileges for Linux. *Computers & Security*, 102983. https://doi.org/10.1016/j.cose.2022.102983

[42] Wazan, A. S., Chadwick, D. W., Venant, R., Laborde, R., & Benzekri, A. (2021). RootAsRole: Towards a Secure Alternative to sudo/su Commands for Home Users and SME Administrators. In A. Jøsang, L. Futcher, & J. Hagen (Eds.), *ICT Systems Security and Privacy Protection* (pp. 196–209). Springer International Publishing.

[43] Samarati, P., & de Vimercati, S. C. (2001). Access Control: Policies, Models, and Mechanisms. In R. Focardi & R. Gorrieri (Eds.), *Foundations of Security Analysis and Design* (pp. 137–196). Springer Berlin Heidelberg.

[44] BPF Compiler Collection (BCC). (2023, May).

[45] BILLOIR, L. ((2023, April). Kapable-clang-sast.

[46] Abedin, M., Nessa, S., Khan, L., & Thuraisingham, B. (2006). Detection and Resolution of Anomalies in Firewall Policy Rules. In E. Damiani & P. Liu (Eds.), *Data and Applications Security XX* (pp. 15–29). Springer. https://doi.org/10.1007/11805588_2

[47] Dennis, J. B., & Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of The Acm*, *9*(3), 143–155. https://doi.org/10.1145/365230.365252

[48] Lampson, B. W. (1974). Protection. *ACM SIGOPS Operating Systems Review*, *8*(1), 18–24. https://doi.org/10.1145/775265.775268

[49] Spence, D., Gross, G., de Laat, C., Farrell, S., Gommans, L. H., Calhoun, P. R., Holdrege, M., de Bruijn, B. W., & Vollbrecht, J. (2000, August). AAA authorization framework. https://doi.org/10.17487/RFC2904