Capability Ambient DO – A Usable Tool for Linux System Security

Capability Ambient DO – A usable tool for Linux System Security

✉ Davide Berardi[1][0000−0002−2473−2439], Luca Bassi[2,3][0009−0005−5191−4309], and Renzo Davoli[4][0000−0002−3377−8594]

[1] Universitas Mercatorum davide.berardi@unimercatorum.it
[2] INFN-CNAF, Bologna, Italy luca.bassi@cnaf.infn.it
[3] GARR, Roma, Italy
[4] Alma Mater Studiorum – Università degli Studi di Bologna renzo@cs.unibo.it

Abstract. POSIX Capabilities are a technique to manage privilege separation in the Linux Kernel, similar to traditional capabilities such as the ones used in microkernels. This approach can create a security mechanism that presents a strong and well-defined privilege separation. Whilst capabilities are a really powerful way to describe user permissions, it can be difficult to analyze the security implications behind every single configuration, making it difficult to not result in a sense of False Security and a cumbersome tuning process. Being a relatively new technology, usable tools to configure Ambient capabilities security subsystems are lacking in the industry scene. To overcome this industrial gap, we propose cado, a usable system to configure capabilities and capability environments. Focusing on usable security and scripting possibilities alongside real-world use-case scenarios, we compare it with reference implementations of similar models. We also present in great detail the attacks analyzed during the creation of the tool, and how cado makes them inapplicable by default in every installation scenario.

Keywords: Linux system Security · Rootless · Capabilities.

1 Introduction

Linux-based Operating Systems are, nowadays, really popular[5] and used in every aspect of computer engineering. The use of these systems ranges from Cloud Computing, Embedded Systems, Communication Technologies Systems such as Virtual Network Functions, and mobile phones (i.e. Android). While monitored and analyzed by experts, these systems are not immune to misconfigurations or tampering from attackers. Recent vulnerabilities, such as the xz-related backdoor, confirm these two hypotheses[6]. To overcome these problems, a correct isolation of privileges is required. How can a company of system and network

[5] As stated by https://gitnux.org/linux-statistics/, at the moment of writing 92.4% of top 1 million servers uses Linux.
[6] Analysis available at https://www.openwall.com/lists/oss-security/2024/03/29/4

administrators create a map of roles, groups, and configurations for each of the management machines and each of its employees? We argue that traditional approaches such as set-user-id binaries and Linux Security Modules (LSM) are not sufficient. Aside from vertical details, we also point out that security systems should be usable, a number of vulnerabilities [1] are due to misconfiguration and complexity of management systems

This Paper is structured as follows:

- In Section 1, Research Questions and the problem we focus on are introduced.
- In Section 2, we present the state of the art, referring to POSIX capabilities, LSM, set-user-id binaries, and the main differences with the proposed approach.
- Finally, in Section 3, we propose a reference implementation of our system, detailing the results and which are the delicate problems we need to take into consideration.
- Sections 4 and 5 highlight the replies to the Research Questions and the related works.

Research Questions. In the following document, we focus on the analysis of the following research questions:

Q1 If the POSIX capabilities model is valuable to secure Linux systems, which are the main threats we could defeat using it?
Q2 Which are the main technological advantages of introducing POSIX capabilities in the security management of a complex general-purpose operating system?
Q3 Which family of software can take advantage of the capability layer?
Q4 How can usability in the daunting task of configuring root privileges be improved?

We argue that replying to these questions can lead to more valuable research in POSIX capabilities, rootless operations, and general-purpose operating system security.

2 State of the Art

The Linux - and more generally UNIX-like - world is slowly moving from having privileged and root-only binaries to rootless operations. Separation of privileges is a delicate matter, with dedicated families of vulnerabilities such as Privilege Escalation and Lateral Movement[2]. Vulnerabilities such as these can lead to the complete submission of the entire machine in case of Escalation to the super-user. To give these problems a numerical value, a discovery of one of these vulnerabilities can be evaluated as much as $50.000 by platforms such as Zerodium[7].

[7] `https://zerodium.com/program.html`, for Local Privilege Escalation (LPE) on Linux as for 25 Apr. 2024.

For instance, Docker - a famous container management tool - recently moved from requiring root to rely only on an unprivileged user on the system[3]. This tactic tends to increase the overall security of the system because the attacker interactions are limited to the ones that can harm only the single user, not the entire machine. Industrial solutions are taking this direction as one of their core concepts. Podman is one of these software focused on the maximization of user separation.

The administrator user of Linux machines, root, historically possesses complete control of the target system. For instance, root is used to change the IP of a network node, read/write the main memory, or mount an external disk. That is, the administrator is an all-or-nothing user, without it a lot of operations are prohibited by default[8], with it, almost everything offered by the system is available. The system also requires a way to switch users, which must be well protected from attackers. This behavior is controlled by a file-system permission called set-user-id (setuid). Executing a program with this bit enabled can give the spawned process the ability to switch to another user. That is the tactic used by *sudo*[4]. While being an effective and simple tactic, it has been proven to be a little bit simplistic and can lead to security problems[5]. This is an obvious violation of the least privilege principle[6]. If your application requires changing the IP of the running machine, it will also inherit the permission to read passwords and so on.

POSIX Capabilities[7] are exactly placed in this part of the security loop. Root privileges are divided into fine-grained permissions. That is, if you need to control the IP-stack-related settings of your operating system, you can designate this capability to a process, without worrying about the fact that it can interact with root-only files in the system.

Linux implements (on kernel version 6.8.7, at the moment of writing) 41 different capabilities, which are described in great detail inside the system manual[9]. These capabilities range from the configuration of the network to the configuration of the file system, security, and so on. This, in conjunction with cgroups and namespaces, completes the isolation of a process.

Management OS-level virtualization such as Docker[8], LXC[8], Systemd-nspawn[9], web browsers sandboxing such as the one used by Chrome, and similar tools use this kind of tools to isolate processes in a limited environment with only the required root configuration possibilities. This is also reflected in the recent updates on software updates and management of Linux-based systems. Above all, Android focuses on great user separation by focusing on containerization. Aside from the extremely customized use of Linux done by Android, distributions such as Ubuntu tend to employ mechanisms such as Snap, a system to distribute sandboxed software. While being certainly a great technological step from the classical model in terms of isolation, privileged programs still need to be executed with extended access to the main system.

[8] In traditional UNIX, a lot of these behaviors are modifiable by using user groups and file system permissions.

[9] The manual is consultable using the command man capabilities.

On the other side of the spectrum, both in scripts and interactive systems, Linux is usually utilized by interacting through a main, unprivileged user. That is, even on single-user machines, no unprivileged operation is performed by root (e.g. develop a script to check for email updates or clone a repository). This approach requires a form of controlled privilege escalation. The reference tool, standard de facto of almost all modern GNU/Linux distributions is Super User Do (sudo)[4]. This tool is usually the most used security-related tool on every Linux system. Due to its popularity and criticality, sudo is heavily overlooked by security analysts. Even with this kind of analysis and overlook, over time security problems arose, creating a class of vulnerabilities with devastating consequences[10,11,12]. When sudo is compromised, due to its implementation, relying on set-user-id permission, the entire root permissions on the system are inherited by the attacker, leading to the complete takeover of the running platform.

To overcome this problem, it is possible to take two approaches: (1) Server Based Mandatory Access Control and (2) the development of a POSIX Capability sudo-like tool. While the first technique performed by Linux Security Modules such as SELinux is validated by scientific literature [13], it adopts an external module (the LSM) to validate the operations performed on the system. The latter one, on the other hand, performs an a-priori tactic, removing the possibility for the user to perform an interaction. The problem with the first approach is the so-called confused deputy problem. LSM and Linux can block, for instance, the access to a predetermined port on the system. If the port can be accessed by, for instance, loading another process and executing it, it can lead to a difficult confinement. Capabilities, on the other hand, are a fine-grained distribution of Root permissions. That means that if a program cannot configure the network, it will never configure the network, without taking into consideration bugs in the Kernel. The absence of confused deputy typical of LSM and the lack of all-or-nothing behavior typical of sudo-like tools make the creation of a sudo-like capability tool the right choice to effectively distribute the root capabilities to administrators with a configuration that is intrinsically compliant with the Least Privilege Principle.

3 Implementation

Before Linux Kernel version 4.3, it was not possible to create a separate subtree of non-root-managed processes that shared a set of capabilities. That is due to the fact that capabilities just relied on the attributes of an executable saved in the file system or dropped by execution system calls. Capabilities are saved as metadata and divided into three sets: Permitted, Inheritable, and Effective.

Upon file execution, these three sets are augmented with two per-thread capability sets, which are called Bounding and Ambient. These five sets have different meanings and control points.

Permitted capabilities are the superior limit to capabilities that are set to the thread that will be executed. If a capability is not listed in the Permitted

set, the thread will never get it, even if it is present in another set saved in the file system. Per-thread capabilities set (which are considered after an execve system call) is not influenced by the Permitted set.

Inheritable capabilities are the set of capabilities preserved when an execve is performed by the application. This is the set that is transferable when a newly executed process takes over the privileged one. This set of capabilities performs in a counter-intuitive way: when the execution of a new process is performed by a non-root user, this capability set is dropped. To maintain retro-compatibility, it is impossible to create sudo-like tools with this kind of capability for unprivileged users.

Effective capabilities are a subset of Permitted capabilities, that contain the capabilities effectively applied by the Kernel. For instance, when an application requests to change the IP address of a network interface, the capability CAP_NET_ADMIN is searched inside this set. If it is present, the change is performed; otherwise, the request is denied, resulting in a permission denied error.

Bounding set, which is the maximum capabilities that can be inherited from the parent and from the capabilities written in the file metadata. That is, this is another limit which is combined between the Inheritable and Permitted sets.

Ambient set, present in the Linux Kernel since version 4.3. This set is preserved through executions of unprivileged processes and is limited by the Permitted and Inheritable set.

As stated in this list, the capabilities set changes upon the call of the execution system request[10]. This is denoted by Equations 1-7.

$$Q(Bounding) = P(Bounding) \tag{1}$$

$$Q(Inheritable) = P(Inheritable) \tag{2}$$

$$Q(Ambient) = \begin{cases} \emptyset, & \text{If executed file is privileged} \\ P(\text{Ambient}), & \text{otherwise} \end{cases} \tag{3}$$

$$K_{\text{Inheritable}} = P(\text{Inheritable}) \cap F(\text{Inheritable}) \tag{4}$$

$$K_{\text{PermBound}} = P(\text{Bounding}) \cap F(\text{Permitted}) \tag{5}$$

$$Q(\text{Permitted}) = Q(\text{Ambient}) \cup K_{\text{PermBound}} \cup K_{\text{Inheritable}} \tag{6}$$

$$Q(\text{Effective}) = \begin{cases} Q(\text{Permitted}), & \text{If } F(\text{Effective}) \neq \emptyset \\ Q(\text{Ambient}), & \text{otherwise.} \end{cases} \tag{7}$$

Where $Q(Set)$ denotes the resultant sets of capabilities after the execution, $P(Set)$ defines the set before the execution of the system call, $F(Set)$ indicates the set of file-system saved capabilities metadata, and K_* are variables to facilitate the execution of the algorithm. As notable, Inheritable and Bounding sets are not modified by the execution. Permitted and Effective sets are at least

[10] The execve system call

configured as Ambient set, even if the Permitted, Bounding, or Inheritable sets are empty. Also, the Ambient set is dropped on privileged applications because the privileged applications are therefore executed with only the pre-configured set of capabilities and not an augmented set.

Having access to this technology offered by the kernel, we implemented the capability management tool, cado, CApability Do, focusing on the user experience and minimal configuration. Being a central security part of the system, this tool requires an attack surface as small as possible.

Our tool is written in C language, with minimal dependencies, this approach also makes cado compatible with most distributions. At the moment of writing, it is available on Debian platforms from its stable edition, Ubuntu, and it is packaged for Arch Linux in the Arch User Repository (AUR) system and it is released as free-software under General Public License (GPL). The source code is available on GitHub at `https://github.com/rd235/cado`. Cado requires a single file, by default placed in a location writable only by the super user (/etc/-cado.conf). This file is therefore read to detect which are the capabilities required for cado to work. This approach is required to create a cado binary compatible with the Least Privilege Principle, cado will be enabled only to manage the specific capabilities configured in the cadoers file. Any successful attack that leads to the compromisation of cado will give the attacker only the capabilities configured inside the file. Let us suppose that we want to isolate the possibility to sniff the network (e.g. by using tcpdump) only to a single user. We have the possibility to do that using a group, but we can perform this control using cado with the following configuration:

```
# echo "net_admin,net_raw: cadouser" >
  /etc/cado.conf
# cado -s
```

This installation procedure gives the user cadouser the possibility to inherit the capability to administer network (CAP_NET_ADMIN) and the capability to manage raw network sockets (CAP_NET_RAW). After this step, the cado binary is reconfigured to have the minimal set of capabilities required (in this case, the ones cited above). This approach is tied to a problem directly related to capabilities. Due to the implementation of the various sets, giving cado a set of capabilities broader than the minimum required will invalidate the Least Privilege Principle. To overcome this problem, an idea is to isolate various cado binaries, one per capability, and limit the execution only to specific users. This approach will honor the Principle, limiting the capabilities used by the tool. While being promising, this approach is still in the development phase and will be analyzed as an extension of this work. After this minimal configuration step, the privilege escalation requiring the capability can be performed in a way that is really similar to the one employed by sudo:

```
$ cado net_raw,net_admin tcpdump -ni any
```

That is, cado limits the required configuration, maximizing the user attention to the security aspects. Cado ensures the authorization of the users to secure

the system from attackers who interact with physically unprotected environments. In the default configuration, cado requires the user password to perform actions. That approach is performed through the standard Linux authentication framework: Pluggable Authentication Module (PAM)[14]. PAM is a heavily configurable approach to manage authentication. It can be easily configured, from allowing every user without any information to requiring complex multi-factor authorizations.

3.1 Script cado

There are cases in which cado can be used in batch scenarios, without requiring password authentication. To overcome this problem in a naïve way, PAM can be configured to never require the user's password.

This would be a big security flaw, every application runnable from the user would be available for an attacker without authentication.

The approach we chose is to create a separate tool to configure the batch mode of cado: scado, Script cado. This tool, in conjunction with cado, enables the use of a particular program or script, without the requirement of a password every time it is required to execute the binary with elevated privileges. To guarantee sufficient elasticity in configuration to reply to Research Question Q4, the system, in our view, must guarantee the following properties:

1. The requirement of an extensible authentication scheme (PAM-compatible);
2. The protection against elevation of modified scripts (offline modification, tampering);
3. The protection against elevation of modified files when running the application (a well known family of race condition attacks called TOCTTOU[15]).

We developed a tool whose workflow can be schematized as follows: (a) the privileged user executes scado -e to start an editor and configure, through a textual file, the privileges that can be assigned to an application. That is, the user will create a file with the following format:

```
/usr/sbin/tcpdump : net_raw,net_admin
```

This configuration file describes that the application /usr/sbin/tcpdump must be configured to automatically inherit the two capabilities listed. The absolute path must be provided; otherwise, tampering can occur (e.g. by executing the script with a different working directory). (b) The privileged user places in a script the command to execute through cado with –scado switch, for instance:

```
#!/bin/sh
set -eu
cado --scado net_raw,net_admin /usr/bin/tcpdump -i any
```

This sh script will execute tcpdump with elevated privileges without the need of authentication. This approach covers just point 1 described before. Users can modify /usr/bin/tcpdump (if they have the right permissions on the file system) and modify its behavior at runtime. To overcome these problems, the configuration of scado can be modified in the following way:

```
/usr/bin/tcpdump : net_raw,net_admin : 326c2365d265b30...
```

That is, a cryptographic hash is appended to the configuration line, describing the signature of the file to execute. When executed, cado checks that the hash of the file is the same as the one intended by the configuration and executes it. This approach was developed to overcome the problem number 2. If the file is modified by the user (without the modification of the relative scado file), the system will detect this modification and will not execute the program with elevated privileges without asking for a form of authentication. Particular actions must be considered for point 3. After checking if the file hash is valid (a specific moment of time that we refer to as "Time of Check"), the attacker can modify it with a different one. If the attacker modified the target before the execution of it by cado (a moment of time we refer to as "Time of Use"), the new file will be executed as valid, in place of the old one. This sequence is an example of the well known attack TOCTTOU (Time Of Check To Time Of Use), a schematic of this attack is presented in Figure 1. This detail, which covers problem 3, is resolved

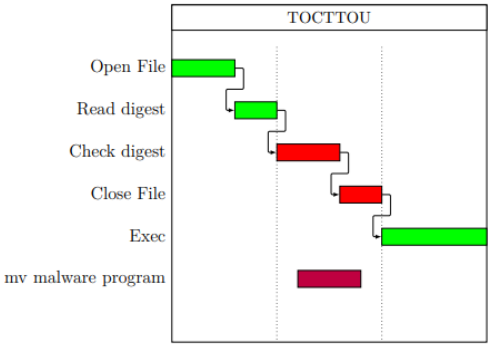

Fig. 1. Time Of Check To Time Of Use attack. The attack window is pictured in red, limited by dashed vertical bars.

by copying the application that must be executed in a private location, before checking and executing it. This approach offloads the protection from TOCTTOU to the confidentiality and integrity provided by Linux file permissions. That is, copying the files into a directory owned by a different user controlled by cado will enforce that the attacker cannot modify the target.

Results. We present the results of cado compared to the software and approaches presented in Section 2 and Section 5. We focus on the real world scenarios implementable and the problems we could manage to find in the other systems.

In Table 1 we present a comparison of metrics we developed to analyze security mechanisms. Some of these metrics are partially applicable to the described tools. For instance, sudo can be compatible with scripts in the so-called "sudo-mode", in which the authentication is granted for a specific window of time, by

Table 1. Comparison between similar software, ○indicates that the problem is absent, ●that the problem is present, and ◑that is present under specific circumstances.

Problem	cado	sudo	LSM	capsh	RootAsRole
If compromised it can lead to the entire system takeover	○	●	●	○	○
Incompatible with scripts (authentication is required)	○	◑	○	●	●
Incompatible with root-less system	○	●	○	○	○
Difficult configuration (i.e. lacking documents, non-sane defaults)	○	○	●	●	●
Not currently adopted by distribution repositories (i.e. can suffer from supply chain attacks)	○	○	○	○	●
Do not honor the Least Privilege Principle	◑	●	●	●	●

default 5 minutes. This is problematic for scripts that need to execute privileged operations in five minutes and workarounds must be put in place. Aside from comparison, we propose a minimal example focused on the configuration of unprivileged web-servers. For instance, if we want to execute a web-server, making it listen on the HTTP well-known port (TCP/80), we would require access to the CAP_NET_BIND_SERVICE capability. To perform this task, we can just issue the following commands:

```
# echo net_bind_service : cadouser > /etc/cado.conf
# cado -s
```

This instructs cado to inherit the required capability and configure it in the capability sets described in 3. The privilege escalation step is therefore straightforward:

```
$ cado net_bind_service -- python3 -m http.server 80
```

The resulting execution of the web-server will be isolated in an unprivileged user, and can be combined with the pre-existing techniques employed by robust web-servers such as Nginx or Apache.

4 Conclusions

Cado and scado are tools that can be used to perform security management of a Linux system. While these tools, at the moment of writing, are significantly less popular than their set-user-id counterparts such as sudo, we argue that they would be a worthy security addition to the Linux ecosystem. In the document, we focused on four research questions that we replied to in the following way:

Q1 In Section 2 we described the main differences between the state of the art of set-user-id-based programs and capability-based ones. We also highlighted

how capabilities are different from LSM and Server-based approaches. This design a specific class of problematics (namely Confused Deputy) that can be defeated by the use of Capabilities.

Q2 Capabilities are difficult to manage, there are different sets of them, and they can be easily foreseen without the use of specific tools or a specific analysis. This leads to a tradeoff where set-user-id-based programs clearly win in terms of configuration but lose in terms of granularity of permissions.

Q3 In Section 3.1, we present a practical use-case in which software such as web servers or specific tools can benefit from the uses of capabilities. We argue that this segregation and segmentation is the basis for a well-secured system, impossible to guarantee without the use of systems equivalent to the one presented in this paper.

Q4 Usable security is a well known topic that combines User Experience and Cyber Security. We described in the work how systems such as all-or-nothing privilege escalation tools are simpler but come with the downside of lower security. In Table 1, we compared the qualitative characteristics of similar tools. We believe that capability-based tools and cado can be a very simple wrap to the complex capability system, providing a very simple method with safe and secure default behaviors.

Future Works: While being distributed and evaluated by the community of main Linux distributions, we are focusing on the improvement of the tool. We are focusing on the creation of a reference implementation of a Linux Distribution completely based on capabilities and by placing cado in a specific slot of a bigger ecosystem. For instance, the V^2 (VirtualSquare) project[16] benefits greatly from the adoption of cado, by being completely integrated in the logic of its virtual machines. Another future work can be focused on the design of the isolation system described in 3. That is, the analysis and implementation of a capability system that honors the least privilege principle to its maximum extent can be perpetrated as an extension of the work.

5 Related Works

Capabilities-based systems are an extensively analyzed topic in the Operating Systems and Cyber Security field. The traditional capability model focuses on the separation of privileges, giving to each process only the allowed set of permissions. These permissions, usually provided as a set of tokens, can be provided when requesting services to the system. Due to this "token-delegation", this approach can be mapped over microkernels or distributed systems. That is, a capability is no more than a token that is provided with every system call and validated by the service provider. This approach is employed by different micro-kernels such as SeL4, Fuchsia, and Fiasco. The extreme approach to this model is the usage of dedicated Hardware to separate privileges. CHERI[17], is a promising approach, focusing on the isolation of memory locations, that can be translated by the system to Kernel objects such as Device Drivers, and system services (e.g. Network Stacks or File Systems). At the moment of writing,

CHERI is available on Field Programmable Gate Array (FPGA) soft-cores emulated RISC-V and ARM platforms and a single board: the ARM Morello. To the best of our knowledge, the only operating systems compatible with CHERI are FreeBSD (in its variant, CheriBSD) and Morello Linux. We argue that this solution is not directly comparable with cado, because of its disruptive approach, which makes it incompatible with most of current and legacy environments. Traditional UNIX does not implement an extended "capability" system, relying on Security checks performed by the Kernel and Access Control Lists. To overcome this problem, solutions such as Capsicum[18], implemented and available in FreeBSD, was developed by Watson et alii. This approach extends the concepts of file descriptors of traditional UNIX systems, giving the application the possibility to interact only inside its subset of possible interactions. Alternatives such as capsh exist. This one, analyzed in 3.1, is developed as a main Kernel tool, without taking into consideration security details or scripting capabilities tackled by cado. The lack of documentation, configuration, and applications makes this tool more prone to human errors, with regard to the approach proposed in this paper. One of the most similar and scientifically accurate works is described by Wazan et alii in [19]. The approach is focused on the configuration of roles and, in conjunction with Role-Based Access Control (RBAC), can make the system secure by mapping capabilities over roles. This approach is overall complex, has a difficult configuration, and, at the moment of writing, does not have a packaged installation, making it more prone to misconfigurations and supply chain attacks.

Acknowledgments. This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Disclosure of Interests. Author Renzo Davoli is a member of committee for TOAST 2024.

References

1. Marchand-Melsom, A., Nguyen Mai, D.B.: Automatic repair of OWASP Top 10 security vulnerabilities: A survey. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. p. 23–30. ICSEW'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3387940.3392200
2. Provos, N., Friedl, M., Honeyman, P.: Preventing Privilege Escalation. In: 12th USENIX Security Symposium (USENIX Security 03). USENIX Association, Washington, D.C. (Aug 2003), https://www.usenix.org/conference/12th-usenix-security-symposium/preventing-privilege-escalation
3. Rahmansyah, R., Suryani, V., Arif Yulianto, F., Hidayah Ab Rahman, N.: Reducing Docker Daemon Attack Surface Using Rootless Mode. In: 2021 International Conference on Software Engineering & Computer Systems and 4th International Conference on Computational Science and Information Management (ICSECS-ICOCSIM). pp. 499–502 (2021). https://doi.org/10.1109/ICSECS52883.2021.00097

4. Binnie, C.: Get Permission from Your Peers with Sudo, pp. 127–134. Apress, Berkeley, CA (2016). https://doi.org/10.1007/978-1-4842-1772-6_13
5. Dmitry, M., Elena, P.: Linux Privilege Increase Threat Analysis. In: 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT). pp. 0579–0581 (2020). https://doi.org/10.1109/USBEREIT48449.2020.9117739
6. Schneider, F.: Least privilege and more [computer security]. IEEE Security & Privacy **1**(5), 55–59 (2003). https://doi.org/10.1109/MSECP.2003.1236236
7. Walli, S.R.: The POSIX family of standards. StandardView **3**(1), 11–17 (mar 1995). https://doi.org/10.1145/210308.210315
8. Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing **1**(3), 81–84 (2014). https://doi.org/10.1109/MCC.2014.51
9. Tolaram, N.: systemd, pp. 325–345. Apress, Berkeley, CA (2023). https://doi.org/10.1007/978-1-4842-8731-6_17
10. MITRE: CVE-2023-7090. CVE Repository (accessed: 25 Apr 2023), https://www.cvedetails.com/cve/CVE-2023-7090/
11. MITRE: CVE-2021-3156. CVE Repository (accessed: 25 Apr 2023), https://www.cvedetails.com/cve/CVE-2021-3156/
12. MITRE: CVE-2019-14287. CVE Repository (accessed: 25 Apr 2023), https://www.cvedetails.com/cve/CVE-2019-14287/
13. Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux security module. NAI Labs Report **1**(43), 139 (2001)
14. Geisshirt, K.: Pluggable authentication modules. Birmingham: Packt Publishing Ltd (2007)
15. Wei, J., Pu, C.: TOCTTOU vulnerabilities in UNIX-Style file systems: An anatomical study. In: 4th USENIX Conference on File and Storage Technologies (FAST 05). USENIX Association, San Francisco, CA (Dec 2005), https://www.usenix.org/conference/fast-05/tocttou-vulnerabilities-unix-style-file-systems-anatomical-study
16. Davoli, R., Goldweber, M.: Virtual square (V2) in computer science education. In: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education. p. 301–305. ITiCSE '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1067445.1067528
17. Woodruff, J., et al.: The CHERI capability model: revisiting RISC in an age of risk. SIGARCH Comput. Archit. News **42**(3), 457–468 (jun 2014). https://doi.org/10.1145/2678373.2665740
18. Watson, R.N., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: Practical capabilities for UNIX. In: 19th USENIX Security Symposium (USENIX Security 10). USENIX Association, Washington, DC (Aug 2010), https://www.usenix.org/conference/usenixsecurity10/capsicum-practical-capabilities-unix
19. Wazan, A.S., Chadwick, D.W., Venant, R., Laborde, R., Benzekri, A.: RootAsRole: Towards a Secure Alternative to sudo/su Commands for Home Users and SME Administrators. In: Jøsang, A., Futcher, L., Hagen, J. (eds.) ICT Systems Security and Privacy Protection. pp. 196–209. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-78120-0_13