**Mini Project Report**

**on**

# Linux Kernel Security

Submitted by

**OM MORENDHA (20BCS095)**

**PRATHAMESH PAI (20BCS103)**

**DIVYANSH MEHTA (20BDS018)**

**POORNIMA (20BDS040)**

Under the guidance of

**Dr. Radhika B S**

**Assistant Professor**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**AND**

**DEPARTMENT OF DATA SCIENCE AND INTELLIGENT SYSTEMS**

**INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DHARWAD**

05/05/2023

## I. Introduction

In the linked world, where we are becoming more and more dependent on technology, such as when systems go online. Additionally, the attack surface for possible weaknesses grows. Linux is not an outlier in this regard; it also has a lot of software vulnerabilities.

Utilizing access restrictions effectively is a crucial step in reducing software vulnerabilities. Resource administrators can limit access in detail using discretionary access control. Each resource in the DAC system has a corresponding Access Control List (ACL), which offers user management and privacy but is insufficient to defend systems from attack. The Linux kernel lacks a default access control scheme, although numerous patches offer improved access controls. The many needs and criteria contribute to the lack of consensus over the access control mechanism to be incorporated into the kernel.

The LSM project minimizes the impact on the Linux kernel while integrating all functional requirements of as many security initiatives as possible. It offers a common loadable module interface for enhancing security. Users have only two options up until recently when intending to set a security policy. Create a custom kernel module or configure an existing LSM module like AppArmor or SELinux. The third and best method for enabling developers to create granular policies without configuring anything or loading the kernel module is LSM-extended Berkeley Packet Filters. Linux computers offer a more complete security solution.

The foundation of the Linux operating system has a number of privilege escalation vulnerabilities that have recently been discovered through the Common Vulnerabilities and Exposures (CVE) system maintained by MITRE. The CVE Scoring methodology gives privilege escalations a high grade because they allow attackers to obtain higher privileges on vulnerable systems. CVE-2022-0185 and CVE-2022-0492 are two such flaws. We are concentrating on Linux kernel privilege escalation vulnerabilities in this paper.

## II. Related Work

In order to make access controls easier, the Linux Security Modules (LSM) project offers a standardized loadable module interface. A security framework must be truly general, conceptually straightforward, least intrusive, effective, and capable of supporting the POSIX.1e capabilities logic, according to Linus Torvalds, who made this clear in his publications. Through the use of hooks in the kernel code placed right before the access, LSM enables modules to mediate access to kernel objects. The module may refuse access when the kernel would grant it, but the module is not considered when the kernel wants to prohibit access. For permissive hooks, where the module can give access that the kernel was about to prohibit, LSM offers some minimal support.

Extensible systems have been the subject of extensive operating systems research over the past 20 years. Modules are completely unrestricted in their access to the kernel's address space thanks to LSM. The only ”restriction” is that the majority of hooks have the ”restrictive” form, which makes it challenging to allow access when it ought to have been prohibited. LSM is mostly reliant on programmer proficiency and root authority (only root is able to load a module). LSM aims to give the Linux kernel all-around support for access control.

BPF has developed from its original purpose as a network packet filter to provide dynamic tracing of the Linux operating system. The BPF language is used to create compact probes that may be injected into the kernel at runtime, enabling flexible and unobtrusive system monitoring. This makes it the perfect tool for security auditing, performance analysis, and issue fixing.

## III. Important Keywords

### A. LSM

[6] A broad kernel framework is offered by the LSM kernel patch to facilitate the inclusion of security modules in the Linux kernel. Access control models can be implemented in Linux as loadable kernel modules, allowing various security policy engine development threads to run independently of the main Linux kernel. This gives one freedom when putting security policies into practice and makes it possible to select the access control solution that best suits the needs.

A standardized interface called LSM makes it possible for security modules to communicate with the Linux kernel. For security modules to employ in enforcing access control policies and other security measures, it defines a set of hooks, or points of control, within the kernel. LSM is compatible with a number of security modules, including SELinux, AppArmor, and TOMOYO Linux. These modules can cooperate with one another to offer a more complete security solution, and they can be loaded into the kernel as LSM plugins.

### B. eBPF

[5] Berkeley Packet Filter (BPF) is a virtual machine and framework used by the Linux kernel to run programs written in the high-level language known as BPF bytecode. System calls, file accesses, network traffic, and other operating system-related activities can all be tracked and analyzed using BPF programs.

LSM enlarged Granular security controls are made possible by the LSM framework's sophisticated Berkeley Packet Filters (eBPF) feature. Security modules have the ability to create BPF programs that may be run in the kernel, making it possible to implement security policies effectively and flexibly without changing the kernel or loading a kernel module. By doing so, the possibility of introducing new vulnerabilities or causing problems with existing kernel modules may be reduced. For a number of use cases, including network monitoring and security, performance analysis, and application tracing, eBPF is created to offer a high-performance, low-overhead solution.

## *C. Namespaces*

In order to give users more control over how resources are utilized and shared, Linux offers namespaces that allow the isolation of certain resources, including process IDs, network interfaces, file systems, and more. For containerization technologies like Docker and LXC, they now serve as the foundation. For instance, the PID namespace isolates the process ID space, allowing each process to have its own set of process IDs that are distinct from those used by processes in other namespaces. When using containerization, when different containers must be kept separate from one another while running on the same host, this can be helpful.

The owner may act as a ”root” inside the USER namespace. It offers a way to separate user and group rights. Restricting processes from accessing resources outside of their assigned namespace, it adds an additional degree of protection. A process is given a special set of user and group IDs that are different from the user and group IDs in the parent namespace when it enters a new user namespace. As a result, processes executing in a user namespace are unable to access resources or carry out operations that demand authorization from a different namespace. User namespaces can be used to manage a variety of system components, including network interfaces, mount points, and interprocess communication (IPC) resources, in addition to separating user and group identities. User namespaces are a potent tool for creating secure and segregated environments in Linux systems as a result of this.

## *D. Container Escape*

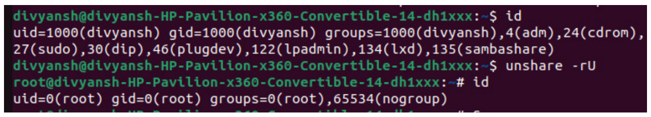
An attacker’s ability to escape a container’s secure environment and reach the host system constitutes a security flaw known as container escape. That amounts to unauthorized access to the host system underneath. This may occur for a number of causes, including incorrect setups, flaws in the kernel or runtime of containers, or mistakes in the container images themselves. Privilege escalation, container breakouts, and network-based attacks are a few typical methods utilized in container escape attacks. Privilege escalation is one method frequently utilized in container escape attacks. This entails taking advantage of flaws in the host system or the container runtime to get elevated privileges that can be exploited to leave the container and access the host system. Container breakouts are another tactic employed in container escape attacks. In order to get out of the container and into the host system, this entails taking use of flaws in the container itself, such as tampering with the file system or vulnerable network interfaces. Container escape attacks may also employ network-based techniques. For instance, a hacker may use a vulnerability in the network stack of the host system or the container runtime to access the network without authorization and carry out additional attacks.

### E. Unshare System Call

[2] A process may ”unshare” its execution context with other system processes using the Linux kernel function unshare(). A separate set of namespaces for the process that are not shared with the parent process or the rest of the system is created when a process calls unshare() to offer more protection and isolation for a specific process or collection of processes. The unshare() system call, which is frequently used in containerization technologies, is a potent tool for building portable and lightweight application environments. due to the fact that it enables containers to establish and maintain their own separate execution contexts, including their own filesystem, network stack, and process namespace.

## IV. Vulnerability

The unshare() system call enables a process to remove one or more namespaces from the parent process and can be used to create a user namespace. A new process with a new user namespace can be created using the clone() system function.



```
divyansh@divyansh-HP-Pavilion-x360-Convertible-14-dh1xxx:~$ id
uid=1000(divyansh) gid=1000(divyansh) groups=1000(divyansh),4(adm),24(cdrom),
27(sudo),30(dip),46(plugdev),122(lpadmin),134(lxd),135(sambashare)
divyansh@divyansh-HP-Pavilion-x360-Convertible-14-dh1xxx:~$ unshare -rU
root@divyansh-HP-Pavilion-x360-Convertible-14-dh1xxx:~# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
```

Fig. 1. Using unshare command to create a new namespace

In figure 1 for the current shell session, the command ”unshare -rU” establishes a new namespace and enters it. In detail, it establishes a new user namespace and links it to the active process. This indicates that the user ID mapping for the current process and all of its children will be distinct within the namespace and separate from the user ID mapping in the parent namespace. This can be seen by using the id command to determine that the user ID mapping for the parent namespace and the new namespace are different.

A process in Linux can unshare one or more namespaces from the parent process by using the unshare() system function, essentially giving the process a new namespace. This can be used to separate processes from the rest of the system or to create a sandboxed environment.

Unshare() is a system call that has occasionally been used by attackers to elevate their privileges on a system. An attacker may be able to obtain elevated privileges, such as the CAPtextunderscore SYStextunderscore ADMIN capability, which enables them to carry out various administrative tasks on the system, by making a new namespace and then using the execve() system call to execute a privileged program inside the namespace.

The terms ”namespace sandboxing” and ”namespace breakout” are occasionally used to describe this method. It can be used to go around security measures and access private data on the system.

The use of unshare() and other system calls should be restricted, least privilege regulations should be in place, and system activity should be closely watched for indications of unauthorized access or privilege escalation in order to prevent this kind of attack.

We will go over our strategy in this report to address the following two namespace sandboxing-related vulnerabilities. Both of these weaknesses have been given high ratings (7 or higher).

### *A. CVE-2022-0185*

[3] This flaw, which affects the filesystem context functionality in Linux kernel versions 5.1-rc1 and up, was found on January 18, 2022. The ”legacy_parse_param” function, specifically, contains a heap-based buffer overflow that makes it possible for an attacker to write outside the boundaries of a kernel memory buffer.

Bypassing any Linux namespace limitations that may be in place, exploitation of this issue could grant an unprivileged attacker root rights. This might provide the attacker access to the compromised machine and let them carry out nefarious deeds. With a score of 7.8, the seriousness of this vulnerability is rated High.

The Linux kernel development team corrected the issue, and on February 4, 2022, the Linux kernel version 5.16.1 was released with the fix.

### *B. CVE-2022-0492*

[4] This flaw, which is a logical error in how cgroups are implemented, could be used by an attacker to escalate their privileges on the system or obtain unauthorized access to system resources. Resource management and distribution among processes are made possible by Cgroups. They are frequently used to limit and distribute system resources among several containers in containerization technologies like Docker and Kubernetes. With a score of 7.0, the vulnerability’s seriousness is graded as High.

## V. Methodology

Privilege escalation is a common attack surface for operating systems. figure 2 explains the workflow of implementing a security module to prevent unauthorized privilege escalation via the unshare syscall in the Linux kernel. The module will use the method to decide whether to approve or deny a certain unshare call. First, we retrieve the call’s current status. Return the outcomes if the previous hooks did not reject the call. Retrieve the task and processor registers in any other situation. The procedures include assessing whether the call is a system call and checking the system’s current state. If the call is not a system call, we will ignore it since it falls outside the purview of our module. If the call is an unshare call, however, we will check for the

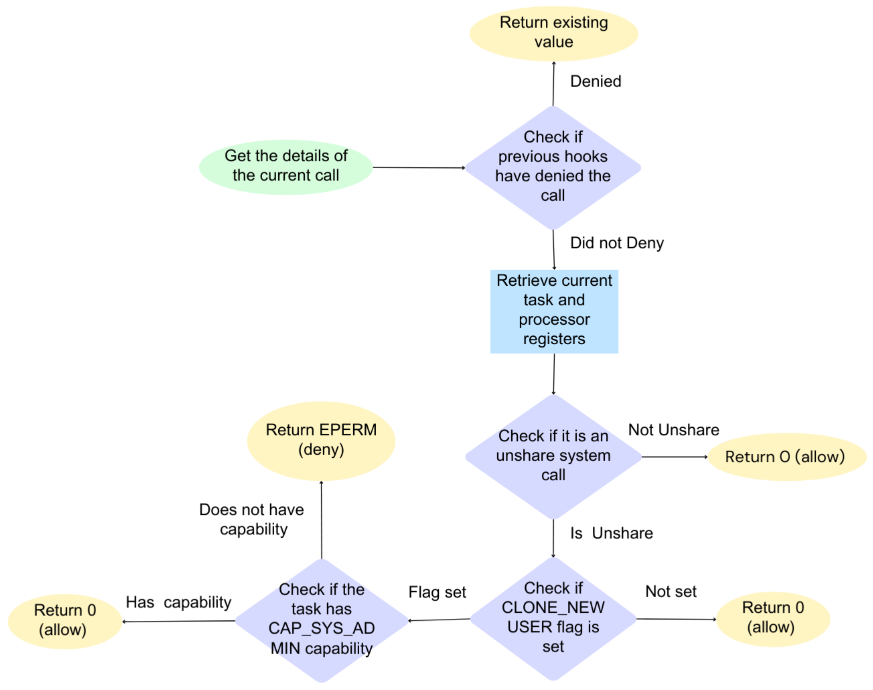
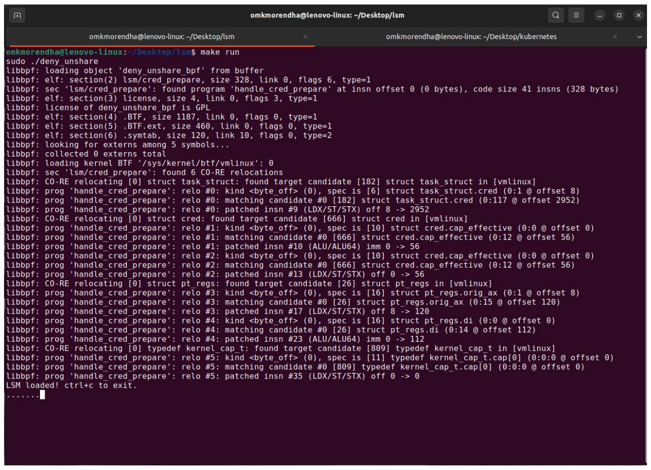

Return existing value
Denied
Get the details of the current call
Check if previous hooks have denied the call
Did not Deny
Retrieve current task and processor registers
Check if it is an unshare system call
Not Unshare
Return 0 (allow)
Is Unshare
Return EPERM (deny)
Does not have capability
Return 0 (allow)
Has capability
Check if the task has CAP_SYS_ADMIN capability
Flag set
Check if CLONE_NEWUSER flag is set
Not set
Return 0 (allow)

Fig. 2. Flowchart of our solution

presence of the CLONE_NEWUSER flag and ignore it if it is absent because there is no danger of a new namespace being created with elevated privileges. Check to see if CAP_SYS_ADMIN capabilities are available for the process. There is a security concern if the module lacks CAP, SYS, and ADMIN capabilities; it would prohibit the operation and produce an EPERM error code. This method is employed to stop unapproved privilege escalation via the unshare syscall.

A user can create a new user namespace with its own UID (user ID), GID (group ID), and capabilities, independent from the rest of the system, by using the unshare system call with the CLONE_NEWUSER flag. Linking the user's new UID and GID to the root namespace enables them to acquire higher rights than they did previously possibly. To stop unauthorized users from utilizing Unshare, it's crucial to have adequate access controls in place.

We need to identify the precise syscall we wish to target and discover suitable hooks to employ in order to prevent privilege escalation attacks using the unshare() syscall. To prevent activities like unshare_userns() that can result in privilege escalation, task-based security hooks like prepare_cred() are frequently used. By doing this, we can stop users who lack the required skills from establishing new namespaces with elevated rights and gaining unauthorized access to confidential information. The git repository contains the code needed to implement this strategy[1].



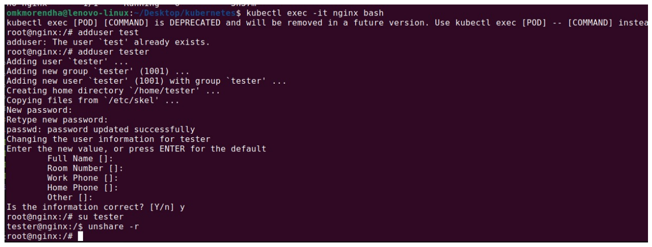
```
omkmorendha@lenovo-linux: ~/Desktop/lsm
omkmorendha@lenovo-linux: ~/Desktop/lsm        omkmorendha@lenovo-linux: ~/Desktop/kubernetes
omkmorendha@lenovo-linux:~/Desktop/lsm$ make run
sudo ./deny_unshare
libbpf: loading object 'deny_unshare_bpf' from buffer
libbpf: elf: section(2) lsm/cred_prepare, size 328, link 0, flags 6, type=1
libbpf: sec 'lsm/cred_prepare': found program 'handle_cred_prepare' at insn offset 0 (0 bytes), code size 41 insns (328 bytes)
libbpf: elf: section(3) license, size 4, link 0, flags 3, type=1
libbpf: license of deny_unshare_bpf is GPL
libbpf: elf: section(4) .BTF, size 1187, link 0, flags 0, type=1
libbpf: elf: section(5) .BTF.ext, size 460, link 0, flags 0, type=1
libbpf: elf: section(6) .symtab, size 120, link 10, flags 0, type=2
libbpf: looking for externs among 5 symbols...
libbpf: collected 0 externs total
libbpf: loading kernel BTF '/sys/kernel/btf/vmlinux': 0
libbpf: sec 'lsm/cred_prepare': found 6 CO-RE relocations
libbpf: CO-RE relocating [0] struct task_struct: found target candidate [182] struct task_struct in [vmlinux]
libbpf: prog 'handle_cred_prepare': relo #0: kind <byte_off> (0), spec is [6] struct task_struct.cred (0:1 @ offset 8)
libbpf: prog 'handle_cred_prepare': relo #0: matching candidate #0 [182] struct task_struct.cred (0:117 @ offset 2952)
libbpf: prog 'handle_cred_prepare': relo #0: patched insn #9 (LDX/ST/STX) off 8 -> 2952
libbpf: CO-RE relocating [0] struct cred: found target candidate [666] struct cred in [vmlinux]
libbpf: prog 'handle_cred_prepare': relo #1: kind <byte_off> (0), spec is [10] struct cred.cap_effective (0:0 @ offset 0)
libbpf: prog 'handle_cred_prepare': relo #1: matching candidate #0 [666] struct cred.cap_effective (0:12 @ offset 56)
libbpf: prog 'handle_cred_prepare': relo #1: patched insn #10 (ALU/ALU64) imm 0 -> 56
libbpf: prog 'handle_cred_prepare': relo #2: kind <byte_off> (0), spec is [10] struct cred.cap_effective (0:0 @ offset 0)
libbpf: prog 'handle_cred_prepare': relo #2: matching candidate #0 [666] struct cred.cap_effective (0:12 @ offset 56)
libbpf: prog 'handle_cred_prepare': relo #2: patched insn #13 (LDX/ST/STX) off 0 -> 56
libbpf: CO-RE relocating [0] struct pt_regs: found target candidate [26] struct pt_regs in [vmlinux]
libbpf: prog 'handle_cred_prepare': relo #3: kind <byte_off> (0), spec is [16] struct pt_regs.orig_ax (0:1 @ offset 8)
libbpf: prog 'handle_cred_prepare': relo #3: matching candidate #0 [26] struct pt_regs.orig_ax (0:15 @ offset 120)
libbpf: prog 'handle_cred_prepare': relo #3: patched insn #17 (LDX/ST/STX) off 8 -> 120
libbpf: prog 'handle_cred_prepare': relo #4: kind <byte_off> (0), spec is [16] struct pt_regs.di (0:0 @ offset 0)
libbpf: prog 'handle_cred_prepare': relo #4: matching candidate #0 [26] struct pt_regs.di (0:14 @ offset 112)
libbpf: prog 'handle_cred_prepare': relo #4: patched insn #23 (ALU/ALU64) imm 0 -> 112
libbpf: CO-RE relocating [0] typedef kernel_cap_t: found target candidate [809] typedef kernel_cap_t in [vmlinux]
libbpf: prog 'handle_cred_prepare': relo #5: kind <byte_off> (0), spec is [11] typedef kernel_cap_t.cap[0] (0:0:0 @ offset 0)
libbpf: prog 'handle_cred_prepare': relo #5: matching candidate #0 [809] typedef kernel_cap_t.cap[0] (0:0:0 @ offset 0)
libbpf: prog 'handle_cred_prepare': relo #5: patched insn #35 (LDX/ST/STX) off 0 -> 0
LSM loaded! ctrl+c to exit.
.......
```

Fig. 3. LSM Running

Figure 3 shows Object 'deny_unshare_bpf' is a bpf program being loaded into the kernel and attached to an LSM hook after being compiled and linked into an object file. Every time the associated LSM hook is called, the kernel will run the BPF program, giving it the ability to carry out additional security checks or reject specific system calls based on the defined logic in the BPF program.

## VI. Results and Discussions

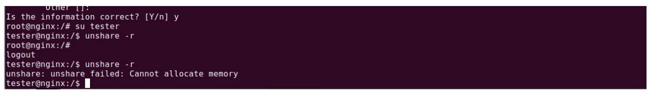
An attacker is able to write the boundaries of the kernel memory's allocated buffer thanks to CVE-2022-0185. Bypassing any namespace limitations in place, a non-privileged attacker can use this vulnerability to elevate their privileges to root. Any user must have at least the current namespace's CAP_SYS_ADMIN privileges. Users can construct or copy namespaces with adequate capacity for unauthorized access using unshare system calls. When using unshare in containerized environments, a pod whose root user does not have the CAP_SYS_ADMIN capabilities can get access. Pods can fail due to container escapes in order to increase privileges. Any pod, deployment, stateful set, replica set, or daemonset is protected by default Seccomp or AppArmor in containerized systems like Kubernetes. We proposed an approach that hunts for uses of unshare utility.



```
omkmorendha@lenovo-linux:~/Desktop/kubernetes$ kubectl exec -it nginx bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl exec [POD] -- [COMMAND] instea
root@nginx:/# adduser test
adduser: The user `test' already exists.
root@nginx:/# adduser tester
Adding user `tester' ...
Adding new group `tester' (1001) ...
Adding new user `tester' (1001) with group `tester' ...
Creating home directory `/home/tester' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for tester
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] y
root@nginx:/# su tester
tester@nginx:/$ unshare -r
root@nginx:/#
```

Fig. 4. Unshare command without LSM

In figure 4 The 'unshare -r' command establishes a new namespace for both the running process and any children it produces. The new namespace is independent of the rest of the system and will not be impacted by any changes to the disc. The child namespace has access to the same file systems as the parent namespace since all mounts from the namespace are mirrored there.



```
        Other []:
Is the information correct? [Y/n] y
root@nginx:/# su tester
tester@nginx:/$ unshare -r
root@nginx:/#
logout
tester@nginx:/$ unshare -r
unshare: unshare failed: Cannot allocate memory
tester@nginx:/$
```

Fig. 5. Unshare command with LSM

Figure 5 shows When using the "unshare -r" command, memory allocation fails because an LSM is present and blocking privilege escalation. After being loaded into the kernel, an LSM module such as "deny_unshare_bpf" intercepts the "unshare" system calls and determines whether the caller has the rights to create a new namespace. The LSM module may produce an error message, such as "cannot allocate memory," if the caller does not have the necessary privileges. This effectively stops the unshare process and avoids privilege escalation.

## VII. Conclusion

An effective method for reducing vulnerabilities is LSM eBPF. By using eBPF, we attempted to resolve a genuine issue. Finding the correct hooks is crucial and requires some attention. It is possible to modify the C-written granular policies to allow specific applications or users to continue using an unprivileged share. To propagate error codes from the cred_preparation hook up the call stack, we suggested a fix.

### A. *Diversity*

The security architecture becomes more diverse when there are several security measures in place to prevent container escape, which might make it more difficult for attackers to identify and take advantage of flaws.

### B. *Customization*

Compared to SELinux or AppArmor, our LSM offers more flexibility and customization choices. Our LSM, for instance, enables more precise control over which individuals or processes are permitted to use unshare.

### C. *Compatibility*

Our LSM can be created to work with particular container technologies or configurations that SELinux or AppArmor do not fully support. For instance, our LSM might be tailored to work best with Docker containers that are powered by a certain Linux distribution.

### D. *Performance*

Comparing our LSM to SELinux or AppArmor, it is possible that it is more effective or has a lower overhead. This may be crucial in contexts where high-performance standards are necessary, such as those for scientific computing or machine learning.

## References

[1] Git repository for the implementation of our approach. https://github.com/omkmorendha/lsm-exp.

[2] The Linux Kernel Archives. Unshared System Call Documentation. https://docs.kernel.org/userspace-api/unshare.html.

[3] NATIONAL VULNERABILITY DATABASE. CVE-2022-0185 from NVD website. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0185, .

[4] NATIONAL VULNERABILITY DATABASE. CVE-2022-0492 from NVD website. https://nvd.nist.gov/vuln/detail/cve-2022-0492, .

[5] eBPF.io authors. eBPF website. https://ebpf.io/.

[6] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *11th USENIX Security Symposium (USENIX Security 02)*, 2002.