Scientific Research Publishing

Journal of Software Engineering and Applications, 2024, 17, 259-269
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Enhanced Memory-Safe Linux Security Modules (eLSMs) for Improving Security of Docker Containers for Data Centers

Juan Martinez Delbugio, Vijay K. Madisetti

School of Cybersecurity & Privacy (SCP), Georgia Institute of Technology, Atlanta, GA, USA
Email: juanmartdelb@gmail.com, Madisetti.vijay@gmail.com

How to cite this paper: Delbugio, J.M. and Madisetti, V.K. (2024) Enhanced Memory-Safe Linux Security Modules (eLSMs) for Improving Security of Docker Containers for Data Centers. *Journal of Software Engineering and Applications*, **17**, 259-269.
https://doi.org/10.4236/jsea.2024.175015

Received: April 23, 2024
Accepted: May 25, 2024
Published: May 28, 2024

Abstract

The adoption of Docker containers has revolutionized software deployment by providing a lightweight and efficient way to isolate applications in data centers. However, securing these containers, especially when handling sensitive data, poses significant challenges. Traditional Linux Security Modules (LSMs) such as SELinux and AppArmor have limitations in providing fine-grained access control to files within containers. This paper presents a novel approach using eBPF (extended Berkeley Packet Filter) to implement a LSM that focuses on file-oriented access control within Docker containers. The module allows the specification of policies that determine which programs can access sensitive files, providing enhanced security without relying solely on the host operating system's major LSM.

Keywords

Docker, LSM, MAC, Rust, Memory Safe Languages

1. Introduction

The concept of Linux containerization, also known as OS-level virtualization, has been around for a long time, as evidenced by technologies such as LXC and OpenVZ. However, its popularity skyrocketed with the introduction of Docker [1] in 2013. Docker revolutionized container management by introducing a simple Dockerfile format for creating Docker images, streamlining the deployment of isolated containers that run software applications.

Docker's easy-to-use approach has led to the widespread adoption of containerization in modern software development. Many applications are now deployed in Docker containers, offering flexibility and scalability. However, with

this convenience come several security challenges. Dockerized container applications may be exposed to malicious attackers both in local networks and the cloud, data center, or the Internet.

These applications often contain multiple programs and sensitive files that interact in complex ways, requiring robust security measures. While efforts have been made to strengthen container security through Mandatory Access Control (MAC) mechanisms such as SELinux and AppArmor, these approaches have limitations. They are often tied to the major Linux Security Module (LSM) of the host operating system and focus primarily on application-based access control rather than protecting critical information assets.

Addressing these limitations and enhancing container security requires innovative approaches that prioritize information assets alongside program-level security measures.

2. Background

In this section, we explore the fundamental aspects of Linux kernel security frameworks and their application to securing Docker containers. We also explore the dynamics of Linux Security Modules, their implementation, and how they can be integrated into the Linux kernel.

2.1. Linux Security Modules

The Linux kernel contains several security subsystems designed to protect both the operating system and the applications running on it. Despite differences in their security objectives, these systems share a common design principle. They use "hooks" built into the kernel to intercept critical security operations (e.g., file access, network-related actions) initiated by processes. These hooks are provided by the kernel in what is known as the Linux Security Module framework, while the implemented subsystems are known as Linux Security Modules (LSMs).

Although the goal of stacking LSMs has been around for over two decades, the Linux kernel supports only one major LSM at a time [2] [3]. This limitation has somewhat limited the development of new LSMs. Currently, SELinux [4] and AppArmor [5] stand out as the most prominent LSMs.

Originally developed and released by the NSA, SELinux has been widely adopted and is pre-installed and enabled by default in distributions such as Red Hat Enterprise Linux and similar (Rocky Linux, AlmaLinux, CentOS Stream, Fedora). It has also influenced SEAndroid, which is the default security framework on Android devices. Consistent with the Bell-LaPadula (BLP) security model, SELinux enforces confidentiality controls through security labels that restrict processes from reading data at higher security levels and writing data at lower levels. However, SELinux can be complicated, making it difficult for novice users to configure.

On the other hand, AppArmor is preferred by distributions such as Ubuntu, Debian, SUSE Linux Enterprise Server, openSUSE, and their derivatives. Unlike

SELinux, AppArmor offers a simpler approach by allowing users to define profiles and associate them with binaries via their paths. These profiles restrict access to files, directories, and resources, providing a level of security. However, AppArmor's focus on binaries may not fit well with file-oriented security approaches that focus on how files can be accessed rather than what applications can do.

2.2. LSMs for Containers

Containers, a form of lightweight virtualization, have been widely adopted for their efficiency and resource utilization benefits. They provide a virtual host environment for applications without the overhead of separate kernels, making them ideal for scenarios that require dense deployment and fast spin-up speeds.

Docker, a leading container platform, leverages Linux kernel features such as kernel namespaces for user isolation and *cgroups* for resource management. However, Docker containers share the same kernel, which limits their access to kernel-level security features available to traditional virtual machines or hosts. In particular, this means that all containers are bound to the major LSM installed in the shared kernel, although efforts have been made to support different LSMs per container [6].

2.3. LSMs in Rust, a Memory-Safe HLL

While originally implemented in assembler, most of today's Linux kernel code is written in C/C++. The choice of C/C++ over other high-level languages (HLLs) is based on performance, direct manipulation of virtual memory, and the absence of a garbage collector (GC). However, there are some drawbacks: HLLs are easier to program, offer many more software abstractions (resulting in simpler code), and, most importantly for security, prevent many classes of memory bugs.

In 2023, according to the CVE database, 190 out of 191 Linux kernel vulnerabilities were related to memory corruption bugs or memory overflows [7]. Not only is the number of vulnerabilities important, but so is the impact. Because operating system code typically runs with the highest privileges, these types of vulnerabilities tend to be critical.

Much research has been done on the use of HLLs for operating system development [8] [9] [10]. In recent years, however, the discussion has become more relevant due to the emergence of Rust [11]. Rust, a multi-paradigm programming language, is notable for its memory management. Using the RAII (Resource Acquisition is Initialization) paradigm, dynamic memory is implicitly freed when the owner variable goes out of scope, preventing many memory-related bugs such as double free, use-after-free, null pointer dereferences, and so on.

While most LSMs continue to be implemented in C/C++, the emergence of Rust provides a compelling avenue for developing new LSMs with improved memory safety and robustness.

2.4. LSMs in eBPF

eBPF (extended Berkeley Packet Filter) is a versatile framework in Linux for creating efficient and secure user-defined programs that run inside the kernel. Applications include network filtering, tracing, and system monitoring.

eBPF programs are dynamically loaded into the kernel and operate in a restricted sandbox environment. However, through this sandbox, eBPF programs can attach to LSM hooks and enhance security and access controls, allowing for custom security policies and monitoring mechanisms.

Despite their limitations compared to major LSMs, eBPF-based LSMs offer unique advantages. They can be dynamically loaded without kernel modifications, and they can complement existing major LSMs by providing additional security measures. For example, SUSE has announced Lockc, an experimental LSM that applies additional measures to limit the capabilities of Docker containers [12].

3. Problem

The widespread deployment of applications in Docker containers in local and cloud-based data centers presents significant security challenges. Accessible from local networks to the broader Internet, these applications become tempting targets for malicious entities. Within these containers, numerous programs interact, often handling highly sensitive files that are critical to the application's functionality.

Managing access control in this context is a fundamental security concern. Traditional access control mechanisms, often based on user permissions and configurations, may not adequately address the specific needs of controlling program access to files and directories.

In addition, traditional LSMs have two main limitations: they are tied to the major LSM enforced by the host operating system, and they have difficulty implementing simple rules to protect highly sensitive files.

This project aims to provide a simpler mechanism for protecting highly sensitive files in containers using mandatory access control policies that specify which programs can access those files, protecting highly sensitive files from being compromised if some of the various components of the Docker container are.

4. Related Approaches

In this section, we will discuss several alternatives for protecting highly sensitive files.

4.1. Traditional Mechanisms

While the traditional Linux mechanism based on user permissions serves as a foundational layer, it often falls short when trying to restrict access based on specific applications.

One option would be to use a different Linux user for each sensitive file. Each

user would own its associated file, and no other user would have permission to access it. Then the programs that are allowed to access that file should be owned by those users and have the SUID flag enabled.

This approach lacks the scalability and simplicity that are fundamental tenets of containerized environments. For example, if a program could access more than one sensitive file, each user should have their own copy of the program, or users should be in the same group, and the binary should use the GUID flag.

Also, this mechanism can be bypassed by gaining root access to the container.

4.2. Major LSMs

Various LSMs provide ways to enforce policies on Docker containers. In particular, Docker provides the—security-opt parameter, which allows users to initiate processes within a container with different SELinux types or AppArmor policies [13].

Developing a custom SELinux policy or AppArmor profile is complicated. While tools like udica [14] for SELinux and bane [15] for AppArmor streamline this process, they focus primarily on container-wide restrictions rather than fine-grained segmentation between applications. Creating nuanced, file-oriented policies requires manual effort and a deep understanding of LSM syntax and transition rules, which is a challenge for many users.

Also, creating policies in LSMs such as SELinux and AppArmor requires high privileges over the kernel and cannot be done from inside the container unless it is configured with high privileges (using the—privileged parameter), a practice that is discouraged due to security concerns and containerization principles. In any case, since the Linux kernel supports only one major LSM at a time, policies are constrained by the LSM running in the host operating system.

4.3. Lockc

Lockc [12] is an eBPF-based LSM written in Rust that is used to restrict Docker containers. While it is compatible with major LSMs, and has the advantages of being developed in Rust and using eBPF, its purpose is not to protect files in containers, but to protect the host operating system from containers.

It is based on the idea that "containers do not contain" and can be used to prevent processes in containers from performing actions such as reading kernel logs, accessing directories that might leak information about the host (/sys/fs, /proc/acpi, etc.), creating bind mounts, and more.

5. Proposed Solution

The proposed solution is an LSM designed to enforce file-based access control policies within Docker containers. This LSM module is able to receive a specified policy file along with the container ID and then apply the specified policy to that container.

The policy file consists of one or more Access Control Lists (ACLs), each of

which specifies a file (by its path) and the programs (by their paths) that are allowed to access the file, along with the permitted access mode.

Here is an example of such a policy (Figure 1).

This example, which is included in the LSM documentation (see Section 10), protects a specific log file by defining that only a specific script can modify it. At the same time, the script is marked as read-only. In this way, the user can guarantee that the log file will only be modified in the way defined by the script.

It's important to note that this policy enforcement works on top of Linux's discretionary access control system. Therefore, the user running the program must still have appropriate permissions for the file, regardless of the access rights of the program.

Files with ACLs defined in the policy are referred to as protected files, while those without ACLs remain unaffected by the LSM. This approach enhances the usability of the module while ensuring that strict access controls are maintained.

Design

The LSM implemented in memory-safe Rust is implemented using eBPF, allowing users to *dynamically* load the module into their operating system and use it alongside the major LSM installed on their system.

The module architecture consists of two main components: a user-space program and an eBPF program, as shown in Figure 2. The user-space program handles user input, including the policy file and container ID, and generates specific data structures used by the eBPF program for access control.

6. Implementation

Since security is a primary concern, Rust was chosen as the programming language for implementation. This decision is in line with the current trend of exploring memory-safe languages for LSM development, as discussed in Section 2.3. The LSM eBPF program was developed using aya-rs [16].

```
// log can only be modified by some script
/var/www/html/log_service/log.txt {
    /var/www/html/log_service/service.php w,
     * r
}

// script is only readable
/var/www/html/log_service/service.php {
     * r
```

Figure 1. Example of a policy.

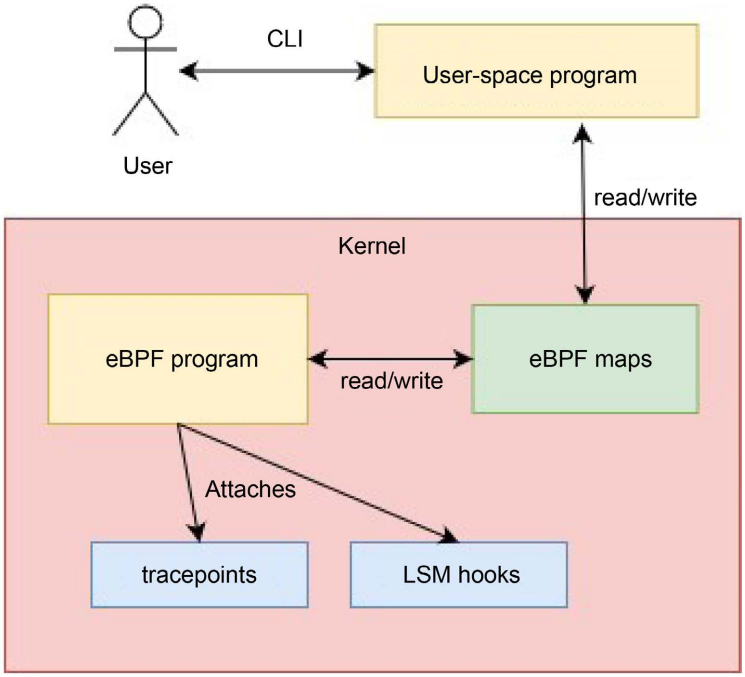

Figure 2. High-level design.

The user-space program is responsible for receiving a policy file and the container ID to which the policy will be applied. It performs tasks such as parsing the policy, retrieving the list of processes running in the Docker container, and loading the eBPF program.

The eBPF program and the user-space program maintain several eBPF maps [17] that allow information to be exchanged between the two components:

- FILE_ID: Maps file paths to numeric IDs. This map is loaded by the user-space program at startup and remains static.
- BINARY_ID: Similar to FILE_ID, but for binary paths.
- ALLOWED_ACCESSES: Maps a pair (file_id, binary_id) to an access mode, indicating permissions for the associated binary over the given file. This map is loaded by the user-space program initially and remains unchanged.
- PID_TO_BINARY: For each **tracked process**, maps the process ID (PID) to the ID of the running binary (or 0 if the binary is unknown). This map is initialized by the user-space program using the process IDs of the Docker container's *cgroup* and is updated by the eBPF program.

To maintain the list of tracked processes (those in the PID_TO_BINARY map), the eBPF program hooks into BTF tracepoints. It removes processes from the map when they exit, and adds new processes when they fork from a tracked process.

The eBPF program also attaches to several LSM hooks. In particular, it detects when tracked processes execute binaries and updates the PID_TO_BINARY map accordingly. In addition, by attaching to file_open and other path-related hooks, the eBPF program detects when a tracked process attempts to access a

protected file and denies the action if the process lacks sufficient permissions.

7. Evaluation

The module has been tested on Ubuntu 22.04 (jammy), but may also work on other operating systems. Also, the Linux kernel version must be higher than 6.8, as this version adds security_path_*-based LSM hooks to the sleepable_lsm_hooks list [18] [19].

For more information about the steps required to run the module, see its documentation (see Section 10). Two examples are documented: the basic example and the web example.

7.1. Basic Example

The basic example (See **Figure 3**) creates a Docker container with a confidential file located in /root/data.txt, and then applies a policy to the container that protects the file to be readable only by cat and vim, and writable only by vim.

Once the module is running, users can verify that no other programs can be used to modify or read the confidential information. It is also impossible to delete, copy, move, change file permissions, or perform any other action that could compromise the confidentiality, integrity, or availability of the file.

7.2. Web Example

The second example (see **Figure 4**) runs a simple web application. The purpose of the web application is to maintain a web-accessible log file (see **Figure 5**) where users can add logs or comments, but cannot modify previous content.

The web application has a command injection vulnerability that allows malicious users to compromise the server. However, by applying a special policy, it is possible to protect the append-only property of the log file.

This is done by creating a script (see **Figure 6**) that encapsulates the semantics of how the log file can be modified (it gets a log and appends it to the file). Then the ACL of the log file specifies that only this script can modify the log file, and

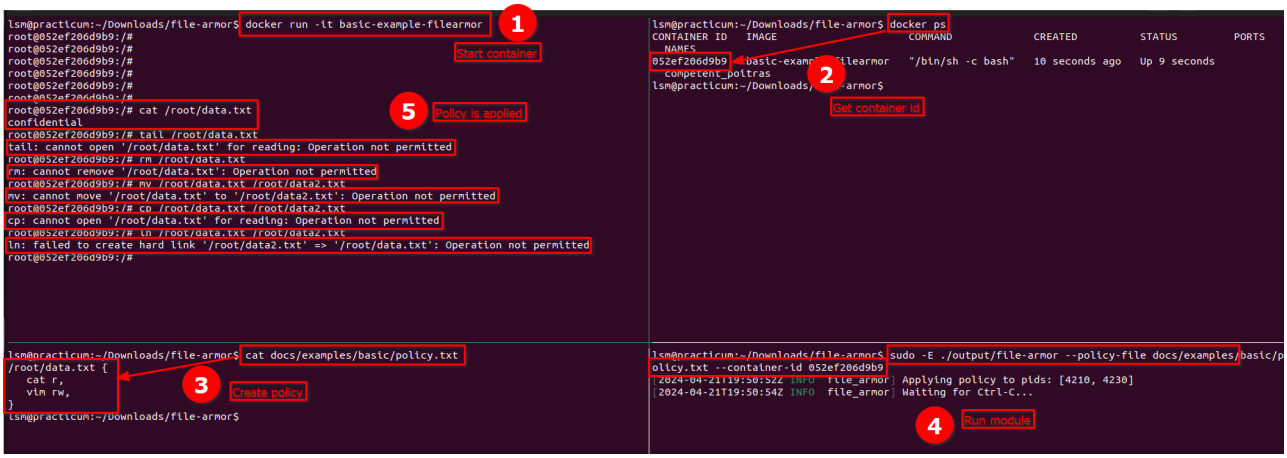

Figure 3. Basic example.

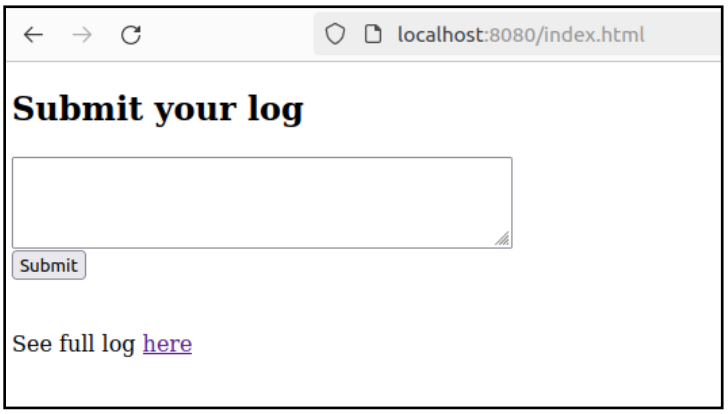

Figure 4. Web example form.

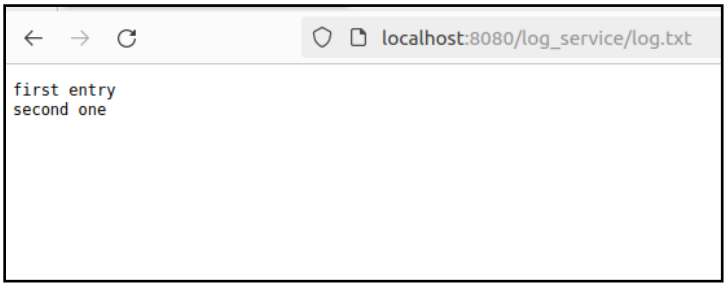

Figure 5. Web example log file.

```php
<?php
$log = $argv[1];
$log_file = dirname(__FILE__) . '/log.txt';
file_put_contents($log_file, $log . PHP_EOL, FILE_APPEND | LOCK_EX);
?>
```

Figure 6. Script that modifies the log file.

the ACL of the script specifies that it cannot be modified by any program.

This approach (using a read-only program that encapsulates the semantics of how to modify the target file) seems to be a good pattern for protecting highly sensitive files, and is a simple solution.

Even after gaining remote code execution on the Docker container, malicious users would not be able to delete the file or change its previous content (they can only append information). This means that even after a full compromise, web application administrators can still recover this information asset.

8. Comparison with Related Projects

The resulting LSM overcomes several limitations of traditional LSMs in Docker environments. First, it can operate independently of the major LSM running on the host operating system. This capability presents eBPF as an alternative for implementing specific LSMs that can be seamlessly loaded into the kernel, enhancing security measures without interfering with the major LSM.

Second, this module simplifies file access control by eliminating the need to define profiles for each application or create unique security labels for each ob-

ject in the system. By applying simple, file-oriented, path-based policies, users can effectively secure their information assets.

Moreover, the module uses Rust, a modern programming language that combines memory safety with high performance. While some LSMs exist in Rust (e.g., Lockc), this solution contributes to the ongoing adoption of Rust in Linux kernel development and signals progress in that direction.

However, the implemented LSM has certain limitations. Currently, the module cannot be applied to multiple containers simultaneously due to static naming conventions for eBPF maps (as discussed in Section 6). This limitation could be overcome by compiling the eBPF program from the user-space application and using unique identifiers for maps in each run.

It's also important to note that the solution developed focuses solely on restricting file access and does not include network-oriented control, inter-process communication, or other types of resource usage.

9. Conclusion

In conclusion, this paper presents a practical and innovative approach to enhancing security in Docker containers through a file-oriented access control LSM implemented using eBPF and memory-safe Rust. The module addresses the limitations of traditional LSMs by providing fine-grained access control to sensitive files within containers. Using Rust for development adds a layer of security and performance optimization. While the module has certain limitations, such as concurrent container support, it represents a significant step forward in securing containerized environments. Future work could focus on addressing these limitations and expanding the scope of control using eBPF and Rust in Linux kernel development, and comparison with existing solutions.

10. Availability

The code is publicly available in the following repository https://github.gatech.edu/jdelbugio3/file-armor, encouraging further research on the topic.

Acknowledgements

We thank the reviewers for their comments that improved the paper.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Merkel, D. (2014) Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2.

[2] Edge, J. (2019) LSM Stacking and the Future. https://lwn.net/Articles/804906/

[3] Corbet, J. (2022) Still Waiting for Stackable Security Modules. https://lwn.net/Articles/912775/

[4] Smalley, S.D., Vance, C. and Slamon, W. (2003) Implementing SELinux as a Linux Security Module.

[5] App Armor. https://apparmor.net

[6] Bacis, E., Mutti, S., Capelli, S. and Paraboschi, S. (2015) DockerPolicyModules: Mandatory Access Control for Docker containers. *2015 IEEE Conference on Communications and Network Security (CNS)*, Florence, 28-30 September 2015, 749-750. https://doi.org/10.1109/CNS.2015.7346917

[7] MITRE Corporation. CVE Linux Kernel Vulnerability Statistics. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

[8] Cutler, C., Kaashoek, M.F. and Morris, R.T. (2018) The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 8-10 Oct 2018, Carslbad, 89-105.

[9] Back, G., *et al.* (2000) Techniques for the Design of Java Operating Systems. *2000 USENIX Annual Technical Conference,* San Diego, 18-23 Jun 2000, 17-20.

[10] Back, G. and Hsieh, W.C. (2005) The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems*, **27**, 583-630. https://doi.org/10.1145/1075382.1075383

[11] MSRC Team (2019) Why Rust for Safe Systems Programming. https://msrc.microsoft.com/blog/2019/07/why-rust-for-safe-systems-programming/

[12] Rosteck, M. (2022) Announcing Lockc: Improving Container Security. https://www.suse.com/c/rancher_blog/announcing-lockc-improving-container-security/

[13] McCune, R. (2023) Container Security Fundamentals Part 5: AppArmor and SELinux. https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-5/

[14] udica-Generate SELinux Policies for Containers! https://github.com/containers/udica

[15] Custom & Better AppArmor Profile Generator for Docker Containers. https://github.com/genuinetools/bane

[16] https://github.com/aya-rs/aya

[17] BPF Maps. https://docs.kernel.org/bpf/maps.html

[18] [PATCH bpf-next] bpf: Add Small Subset of SECURITY_PATH Hooks to BPF sleepable_lsm_hooks list. https://lore.kernel.org/all/ZXM3IHHXpNY9y82a@google.com/

[19] bpf_lsm.c-kernel/bpf/bpf_lsm.c-Linux Source Code (v6.8)-Bootlinmaps. https://elixir.bootlin.com/linux/v6.8/source/kernel/bpf/bpf_lsm.c