Landlock: From a security mechanism idea to a widely available implementation

Mickaël Salaün
mic@digikod.net

Microsoft

Abstract. Landlock's goal is to make it possible for Linux applications to sandbox themselves. On Linux, many traditional access control mechanisms are only available to the system administrator, which do not follow the principle of least privilege. As a result, sandboxing policies were created independently of an actual program execution, leading to unnecessarily broad policies. With Landlock, unprivileged processes can safely create sandboxing policies well-tailored to the expected needs of a running application. Landlock also solves the organizational aspect of keeping policy and software in sync with each other, by putting the policy definition and maintenance in the developer's hands.

The development of Landlock happened in three steps: design, integration in the Linux kernel, adoption by distributions and developers. This article gives our feedback on all these steps, which are all crucial to widely protect users.

1 Introduction and goal

Linux is used globally across various applications, including end user devices and cloud computing. Most security features are focused on system administrators and distribution maintainers, excluding a subset of users such as developers or end users. Even if the rich application ecosystem (e.g., developer tools, network services, smartphone apps) is a major reason for the success of Linux, no security features were dedicated to confine applications (i.e. sandboxing) and protect all users.

The goal is to improve the security of all Linux users and especially to protect users from attacks targeting their applications. It is assumed that all initially trusted software can become malicious once compromised, which is the motivation to isolate those components from one another. This led us to develop Landlock which brings a suitable sandboxing mechanism to Linux. Landlock empowers developers by putting security policy creation and maintenance in their hands, closer to the programs that the policy is about. Because all users rely on programs, Landlock can empower everyone. This primitive leads to useful development properties and security guarantees explained in this article.

2 Properties of a security sandboxing mechanism

One of the initial threats for multi-user operating systems was one user accessing data from another user. This leads to defining security policies to protect users against each other. This kind of policy (i.e. Mandatory Access Control) must then be defined by an entity with greater privileges, either the system administrator or the distribution developer. Sandboxing is different because it is available to everyone.

2.1 What is sandboxing?

A sandbox is defined as "a restricted, controlled execution environment that prevents potentially malicious software [. . .] from accessing any system resources except those for which the software is authorized" [22].

Users can manage data for different use cases (e.g., one per customer) with a set of applications. An application instance can get compromised and act against its user by accessing data on their behalf. The goal of sandboxing is then to isolate attacks in sandboxes the same way users are isolated from one another.

Application development and distribution models bring additional constraints. Frequently, the same application runs on a whole set of supported systems (e.g., all Linux distributions). A sandbox mechanism needs to be flexible enough to protect users as much as possible according to the running system.

2.2 Dynamic policy composition

Any process should be able to sandbox itself, which means that the system should be able to compose hierarchically nested security policies, which are aligned with the hierarchy of processes and their parent relationships, but also taking into account any other access control mechanisms enforced by the system.

Because each user can launch applications several times, each of them must be able to sandbox themselves, creating sibling sandboxes. This means that each application defines its own security policy, and all of them must be enforced by the kernel. This also means that the lifetime of such a policy is tied to the set of processes being restricted.

Through nested sandboxing, an environment or an application can restrict itself further. For instance, the init system might create a sandbox for itself, then for the user session, then the user might sandbox a shell and launch an application with embedded sandboxing. To get an efficient and then usable access control, handling nested sandboxes must not lead to a significant performance impact.

In figure 1, the composition of all security policies enforced on a system may include sibling and nested sandboxes. The kernel needs to manage this set of ephemeral policies in a consistent way while the system is running.

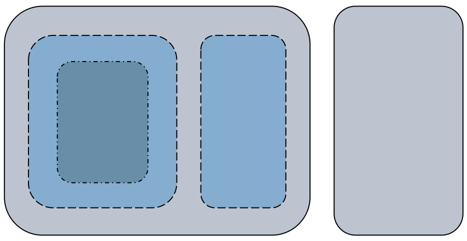


Fig. 1. Composition of isolations: nested and sibling sandboxes

2.3 Principle of least privilege

A sandboxing mechanism available to application developers implies providing an access control mechanism safe for unprivileged users. Applying the principle of least privilege to sandboxing, dropping privileges should be an unprivileged operation (e.g., not relying on a SUID binary nor a privileged service).

In practice, this means that Linux capabilities should not be a requirement. Any privilege elevation mechanism such as *setuid* binaries should not be required, and they should even be denied so that they cannot be abused to bypass the sandbox [5]. Similarly, relying on a user space security broker leveraging privileged features (e.g., filter and forward system calls) put the security guarantees on both the kernel and the broker. Because this broker's goal is to restrict the use of some kernel resources (e.g., files) for another kernel resource (i.e. a process), incorrect mirroring of the kernel semantic and state, or race conditions, could create vulnerabilities [14]. Anyway, trying to shoehorn a privileged security mechanism into an unprivileged one should raise a red flag.

2.4 Innocuous access control

Being able to enforce restrictions on ourselves should not lead to privilege escalation, which could be caused by denying access to resources for more privileged subjects (i.e. processes). A first limitation should then be that a subject should only be able to enforce restrictions on itself, or on equally or less privileged subjects.

A simple way to apply this principle would be to tie restrictions to a scope of subjects, including the requester. However, we need to be careful and consider different kinds of confused-deputy attacks.

2.5 Protecting against bypasses

The sandboxing mechanism implementation should make sure that there is no way to bypass enforced security policies.

Impersonation A sandboxed subject must be confined from less sandboxed ones. It must not be allowed to impersonate a more privileged subject, which could lead to privilege escalation and then a policy bypass. In addition, it must not be allowed to access data from subjects with more privileges, which could include data not otherwise accessible (e.g., file's content copied in memory) and bypass integrity or confidentiality.

Access-control consistency A sandboxed subject must not be allowed to perform confused-deputy attacks on more privileged subjects (regarding restricted resources). For instance, it must not be possible to pass a resource with a restricted set of rights (e.g., file descriptor) to a more privileged subject and get back the initial resource with more rights.

Policy correctness When implementing a sandboxing mechanism, it might not be possible to enforce these principles for all more privileged components. For instance, a kernel can only let user space know about less privileged processes but not make sure that they correctly request this information nor correctly take it into account. A security policy misconfiguration could allow a sandboxed process to tamper with its own or other's persistent data (e.g., configuration file, cache) and then alter behavior of next runs (e.g., disable sandboxing), leading to privilege escalations. The enforcing component (e.g., kernel) may not be able to protect nor warn against this kind of security policy misconfiguration because it may not know about these sensitive data.

3 State of the art of sandboxing in non-Linux systems

Most general-purpose operating systems provide at least a way to configure an access control system, and some of them are sandboxing mechanisms.

3.1 XNU Sandbox

XNU Sandbox [4], previously called Seatbelt, is a security component used by iOS and macOS. It is based on the TrustedBSD security framework [42]. Filesystem and network access rules are defined with an *S-expression* language, and files are identified by path via regular expressions.

3.2 Pledge and Unveil

The `pledge()` [23] system call is a sandboxing mechanism developed and used by OpenBSD. It enables us to define a set of allowed accesses with *promises*, each covering a set of related system calls. For instance, the `dns` *promise* allows DNS network transactions. The `pledge()` implementation makes assumptions about OpenBSD's file topology (e.g., hard coded `/etc/resolv.conf` path).

The `unveil()` [24] system call complements `pledge()` for file path restrictions. It enables us to define a set of file hierarchies for which a given set of actions are allowed: read, write, execute, and create.

3.3 Capsicum

The goal of Capsicum [43] is to bring the capability principle to UNIX systems, currently FreeBSD, using file descriptors to pass capabilities. The capabilities are a way for compatible applications to finely expose their resources with a set of allowed access to other processes.

The main drawback is that this mechanism often requires bigger changes to an application's design. It might also be required to rely on a set of brokers like Casper [13] to control sensitive actions.

3.4 AppContainer

Windows's AppContainer [18] enables us to isolate execution environments. This includes credential, device, file, registry, network, process, and window isolations.

4 Linux security features

Table 2 compares different Linux mechanisms that may be used for some kind of sandboxing.

Using a Virtual machine (VM) to protect the host from the guest makes sense if few of the host's resources (e.g., files, scheduling, IPC) are shared with the guest. The host running the VM cannot reason why the shared resources are being used because another independent operating system is managing the contents of the VM (e.g., guest's files, scheduling, IPC). Moreover, running a VM requires privileges on the host, and embedding it in an application would require a complex and integrated mechanism such as Application Guard [19], which does not fit to the kernel's realm. Here, we are looking for a sandboxing mechanism able to control kernel resources.

The Linux kernel provides complementary security mechanisms, including several access control systems implemented as Linux Security Modules (LSMs): SELinux, AppArmor, Smack, and Tomoyo. However, they are designed to be configured by the system administrator and then defined as Mandatory Access Control (MAC).

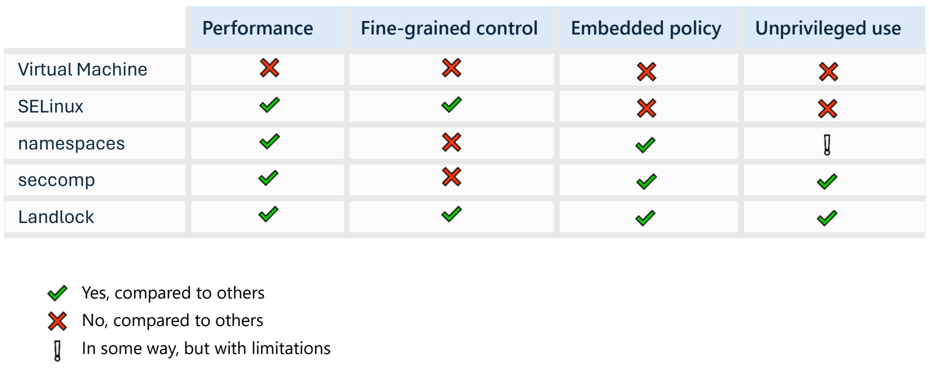
Namespaces are the main building blocks of containers. [1] They are designed to create views of the kernel resources, but not to enforce access control policies. Moreover, they are designed to be used by privileged users (e.g., *root*) and giving access to such power to unprivileged processes can lead to privilege escalation. Things are changing a bit with user namespaces, but there are still ongoing security issues tied to such complexity [6, 10].

seccomp was designed to protect the kernel from malicious user space processes. It works like a firewall for system calls, and can then filter them according to their raw arguments. Some new features also enable user space to emulate system calls, which is useful for compatibility fixes. However, *seccomp* is not an access control system and cannot filter system calls according to the underlying kernel object semantic (e.g., file, socket). *seccomp*'s API is very powerful but also very complex because of filters being BPF programs, which is an important practical concern for adoption. Moreover, because of new or updated syscalls, there might be compatibility issues across kernel versions, architectures, and *libc* changes [7].

Landlock was designed to fill the need for a sandboxing mechanism on Linux and meet all these goals:

— low overhead whatever the number of (nested) sandboxes;

[1] *cgroups* can also be used by containers to restrict processes, but they are mostly designed to limit resource usage.



	Performance	Fine-grained control	Embedded policy	Unprivileged use
Virtual Machine	✗	✗	✗	✗
SELinux	✓	✓	✗	✗
namespaces	✓	✗	✓	!
seccomp	✓	✗	✓	✓
Landlock	✓	✓	✓	✓

✓	Yes, compared to others
✗	No, compared to others
!	In some way, but with limitations

Fig. 2. Comparison of different access control mechanisms

— fine-grained access control system for files, network, and other kernel resources;
— security policy embeddable in applications, with all related challenges (e.g., policy composition, simple-enough and flexible API);
— usable by any processes, privileged and unprivileged.

5 Design constraints and principles

Applications are built on top of the system ABI, of which an important and critical part is the kernel ABI. In a nutshell, the goal of the kernel is mainly to share hardware resources with processes of different trust levels. To maintain the required security boundaries, the kernel implements a set of access control mechanisms (e.g., DAC, MAC). We want to extend these mechanisms with a new one to support the sandboxing approach.

5.1 Kernel ABI

The main contract between the Linux kernel and user space is a low level Application Binary Interface (ABI) provided through system calls.

Some synthetic filesystems (e.g., `/proc`) and special files might be accessible to an application, but the kernel cannot make any assumption about the contents and usage of files. For instance, this is not the case with the OpenBSD kernel that can rely on more assumptions about user space and file topologies (see section 3.2).

5.2 Kernel flavors

In the general case, applications cannot assume that all required kernel features are available. Because generic Linux distributions allow great flexibility, system administrators can select a specific kernel version while keeping user space as is.

Moreover, because the Linux kernel is highly configurable (at build and run time), the available features can vary even within a single kernel version. For instance, almost all Linux distributions build their own kernel with their own configuration.

Respecting all supported versions, this gives a glimpse of the wide variety of running kernels in the wild. Linux's flexibility is powerful, but it comes at a cost in maintenance and compatibility. When developing a new kernel feature, it is needed to consider the variety of supported kernels for wide adoption.

5.3 Multiple interfaces

Kernel resources can be identified and used in different ways. For instance, a file descriptor can be acquired not only through the `open` system call but also passed through a unix socket.

5.4 Sensitive kernel changes

Every change, including new security features, to a privileged component such as the Linux kernel, is a risk of introducing new (security) issues.

One of the goals of the kernel is to define exposed resources with well-defined semantics and related access controls. The kernel needs to be modified to extend the way these access controls are configured.

The kernel community mitigates these risks through code reviews, design reviews, and tests to define guarantees and make sure they are kept over time. Moreover, the implementation can be assessed by anyone (and it happens in Linux), which can then improve our trust in this component.

5.5 Security policy principles

Landlock's design and implementation follows a set of guiding principles to avoid classes of implementation issues for sandboxes.

Access control is expressed with kernel objects (e.g., file, process) instead of system call filtering (i.e. syscall arguments) unlike *seccomp*.

A security policy cannot define the error codes returned by system calls (e.g., `EPERM`, `EACCES`, `EXDEV`) nor change the kernel interface semantic to avoid compatibility issues (see section 7.3).

To protect against multiple kinds of side-channel attacks leading to parent policy leaks, kernel data leaks, or access requests leaks, the security policy is not programmable (attacks based on execution time, speculative execution time, or data access time) nor can communicate with user space (e.g., using eBPF and the related maps). Furthermore, relying on a program to define a layer of sandboxing would make the security policy composition much more complex and it would have a high impact on performance (see section 7.5).

Sandboxing operations such as defining or enforcing a security policy only tax processes requesting such sandboxing for the required resource usage (e.g., CPU usage, kernel memory allocations). In addition, the implementation of kernel access checks does not have a noticeable performance impact on unsandboxed processes. Only sandboxed processes may notice a small performance impact, especially when requesting a denied action. These principles are crucial to scope the sandboxing constraints to only a set of restricted processes, and to protect the system against denial of service.

6 Landlock interface

Landlock is used through a set of user space interfaces to define sandbox security policies.

6.1 Access rights

As for UNIX file access checks, most access rights are checked at file open time, which limits the performance impact throughout the lifetime of the resulting file descriptor. Sets of file access rights are defined as a bit mask where individual access rights are named starting with with `LANDLOCK_ACCESS_FS_` These rights can be applied to files to control executability, readability, and mutability: `EXECUTE`, `READ_FILE`, `WRITE_FILE`, `TRUNCATE`. Another set of directory entry access rights can be applied to control visibility and creation: `READ_DIR`, `REMOVE_DIR`, `REMOVE_FILE`, `MAKE_CHAR`, `MAKE_DIR`, `MAKE_REG`, `MAKE_SOCK`, `MAKE_FIFO`, `MAKE_BLOCK`, `MAKE_SYM`, and `REFER`. Being able to differentiate between file types is useful to easily control creation of new interfaces (e.g., socket, character device).

Network access rights are prefixed with `LANDLOCK_ACCESS_NET_` and currently control TCP `bind` and `connect` actions with `BIND_TCP` and `CONNECT_TCP`. Each access right is explained in the official documentation [33].

6.2 Landlock rules, rulesets, and domains

Landlock is a deny-by-default access control, but with a fixed set of access rights for compatibility reasons. A Landlock ruleset defines a security policy provided by a process to sandbox itself. Each ruleset handles a set of restrictions, and additional rules can add exceptions to these constraints (i.e. allow-list approach). When restricting a process, the kernel merges the process's inherited security policies, called a Landlock domain, with the provided ruleset to create a new domain, composing all these restrictions. Each Landlock domain is tied to at least one process, and the domain ends when the last process exits.[2]

In the example of figure 3, an unsandboxed process P1 can spawn a first child P2, and then sandboxes itself before creating a second process P3. In this case, P1 and P3 are sandboxed by the same initial (red) domain, but P2 is still unsandboxed because it was created before the sandboxing. Then, when P3 sandboxes itself and spawns its first child P4, they are both restricted by the new (green) domain but also by the parent (red) domain.

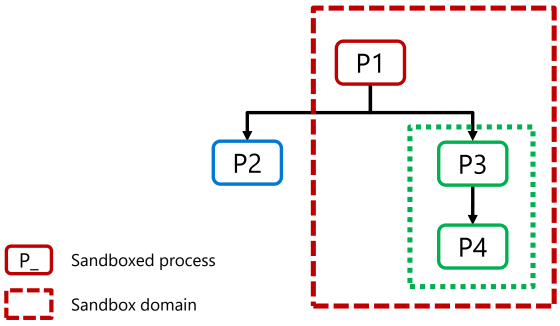


Fig. 3. Sandbox hierarchies

6.3 Compatibility properties

To limit the cost of review and the impact of potential issues, Landlock started as a Minimal Viable Product (MVP). Landlock is now gaining more features over time, including new access rights.

Because of compatibility reasons, previous features need to be supported, and only new ones are added. The second version of the Landlock

[2] Multithreading is a special case discussed in section 6.5.

ABI added support for file reparenting (see section 7.3). The third version added support for file truncation. The fourth version added initial networking control for TCP `bind` and `connect`.

Because Landlock is not a fully featured access control system yet, application requirements that might not be fulfilled by a specific kernel version need to be considered. An application should not sandbox itself if there is a risk to break a legitimate use case.

Compatibility between the kernel and user space is important for common features, and more important for security features. Indeed, being able to use the available kernel security features as much as possible is crucial if the goal is to protect users as much as possible. Application developers cannot expect all their users to use the same kernel version (see section 5.1), and thus should follow a best-effort security approach: leverage all available Landlock features without failing because of unsupported ones [21]. However, this may be constrained by some minimally required access and it may be complex to implement correctly.

Landlock is designed to be as simple as possible in the kernel side, and to move the compatibility complexity to user space. For instance, the Rust library is designed to make it easy for developers to use while providing guarantees that their users will be protected as much as possible [35].

6.4 Policy definition suitable for embedded sandboxing

Landlock tackles the problem of application-defined sandboxing, which means embedding a security policy into an application (i.e. built-in policy), as close as possible to its semantic. A very useful property of embedded sandboxing is that there is no need for an explicit security policy defined by users because such policy can be implicit due to the application's configuration and requirements. Of course, implementing a sandboxing mechanism with the related constraints means that Landlock also supports many more use cases requiring fewer constraints such as for any process spawning another application (e.g., sandbox manager, init system, shell). Being able to enforce complementary layers of security according to different trust levels is also a very important property (see figure 7 explained in section 7.3).

Being able to easily integrate a security policy in an application also brings some very useful properties such as testing. Indeed, a standalone security API enables us to test security features as close as possible to the business logic, similarly to other features. Good development hygiene such as continuous integration tests makes it possible to have security guarantees about the embedded sandboxing, but more importantly to make sure that all functional tests run well with such restrictions [35].

To make it possible to embed a security policy in any Linux application, among all kernel interfaces, it can only be assumed that system calls are available.[3] Indeed, synthetic file systems may not be visible (e.g., containers), and more generally other access control systems may already be enforced and limiting the available interfaces. This led us to implement new system calls dedicated to Landlock.

6.5 Landlock system calls

Landlock provides three system calls: `landlock_create_ruleset()`, `landlock_add_rule()`, and `landlock_restrict_self()`. Following the builder pattern, the first syscall creates a ruleset, the second syscall populates the ruleset, and the third syscall enforces the ruleset. This API is very flexible and was designed to easily add new features to Landlock while still being compatible with the previous ones.

Unlike OpenBSD's `pledge()` syscall which takes strings as argument, Landlock syscalls take bitflags, enums, pointers to specific types, sizes, or file descriptors. Landlock's interface is more generic, which is required because Linux user space is versioned and installed independently from the kernel. Backward compatibility at the syscall layer must then be guaranteed. For instance, most Linux distributions provide several kernels, but the same set of user space components. Another benefit of Landlock's approach is to be able to get help from tools such as compilers or linters, which could not help with strings as function arguments to check consistency and compatibility. Manipulating bitflags and appropriate system calls also makes it easier to programmatically create Landlock rules.

Unlike *seccomp*, Landlock does not filter system calls, but controls access to kernel resources (e.g., file, process). Therefore, Landlock's rules do not need to be kept in sync with new Linux system calls because the kernel's semantic is maintained with the LSM framework. *seccomp* was designed to protect the kernel and, as such, remains very valuable.

`landlock_create_ruleset(attr, size, flags)` This system call creates a new ruleset.

The `attr` argument is a pointer to a `struct landlock_ruleset_attr` defining a set of access rights denied by default. This struct is extensible and will gain new fields to define more access types.

[3] As explained in section 9.1, that may not even be the case in practice, but the syscall interface is still the best choice for standalone features.

The `size` argument lets user space declare the size of the ruleset attributes. Because newer kernels will handle more attributes, which will make this struct grow, the kernel needs to know the number of provided attributes. The compatibility trick is for the kernel to accept any trailing zero values up to a maximal size. This way, user space can update the `struct landlock_ruleset_attr` type with a larger one, and it will remain compatible as long as the new fields are not set. If the kernel receives unknown fields (i.e. trailing non-zero values), the `E2BIG` error code is returned.

The `flags` argument, as for any other Landlock syscall, is an optional flag. This is a good design to leave room for future features. `LANDLOCK_CREATE_RULESET_VERSION` is the only valid flag for `landlock_create_ruleset()`. It is used to get the Landlock ABI version of the running kernel as an integer. As explained in section 6.3, leveraging the documentation or a Landlock library, it is then possible to know all available features. This makes it possible to adjust the security policy according to the kernel capabilities. This design was preferred because it is the simplest one, which reduces kernel complexity while enabling user space to infer all required information. Any new Landlock feature must increment this version and update the documentation accordingly.

In listing 1, the `ruleset_attr` variable is initialized with the handled file access rights, which will all be denied unless explicitly allowed by a rule. If the call to `landlock_create_ruleset()` failed, then a negative error code is returned, otherwise a ruleset file descriptor is returned. This file descriptor identifies the newly created Landlock ruleset and makes it possible to change or use it.

Listing 1: Create a Landlock ruleset

```c
struct landlock_ruleset_attr ruleset_attr = {
        .handled_access_fs =
                LANDLOCK_ACCESS_FS_EXECUTE |
                LANDLOCK_ACCESS_FS_WRITE_FILE |
                ... |
                LANDLOCK_ACCESS_FS_MAKE_REG,
};

int ruleset_fd = landlock_create_ruleset(&ruleset_attr,
↪   sizeof(ruleset_attr), 0);
if (ruleset_fd < 0)
        error_exit("Failed to create a ruleset");
```

`landlock_add_rule(ruleset_fd, rule_type, rule_attr, flags)`
This system call populates a Landlock ruleset with a new rule.

The `ruleset_fd` argument is the file descriptor identifying the ruleset to add an exception to.

The `rule_type` argument is an `enum` identifying the type of the rule: `LANDLOCK_RULE_PATH_BENEATH` or `LANDLOCK_RULE_NET_PORT`.

The `rule_attr` argument is a pointer to a rule structure as defined by the second argument. When `rule_type` is set to `LANDLOCK_RULE_PATH_BENEATH`, `rule_attr` should be a pointer to a `struct landlock_path_beneath_attr` value identifying a set of accesses for a file hierarchy expressed by a (parent) file descriptor. Similarly, if `rule_type` is set to `LANDLOCK_RULE_NET_PORT`, the rule would be defined with a `struct landlock_net_port_attr` value identifying a set of accesses for a network port.

In listing 2, a `path_beneath` variable defines the rule allowing a set of accesses on the `/usr` file hierarchy. If the call to `landlock_add_rule()` succeeds, then the Landlock ruleset was correctly updated.

Listing 2: Add a new Landlock rule to the ruleset

```c
int usr_fd = open("/usr", O_PATH | O_CLOEXEC);
if (usr_fd < 0)
        error_exit("Failed to open file");

struct landlock_path_beneath_attr path_beneath = {
        .allowed_access = LANDLOCK_ACCESS_FS_EXECUTE | ...,
        .parent_fd = usr_fd,
};

int err = landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH,
↪   &path_beneath, 0);
if (err)
        error_exit("Failed to update ruleset");

close(usr_fd);
```

`landlock_restrict_self(ruleset_fd, flags)` This system call restricts the calling thread with the ruleset identified by `ruleset_fd`.

The Linux kernel manages task's credentials (e.g., UIDs, capabilities) per thread. This does not mean that restricting a thread with Landlock or anything else would give any security guarantee. Threads are mostly units of execution, sharing resources (e.g., memory, file descriptors) with all threads from the same process. Therefore, any thread can tamper with the memory used by sibling threads, and then control their execution. This means that threads should not be used as security boundaries. However, being able to manage credentials per thread gives some flexibility that can be used to create safeguards or tests. A process sandboxing itself should then make sure that the calling thread is the only thread from this process.[4]

In listing 3, to avoid privilege escalation by executing a SUID binary and restricting it, a thread must first call `prctl` with the `PR_SET_NO_NEW_PRIVS`. If this is not done, any `landlock_restrict_self()` call will fail.[5] `landlock_restrict_self()` can then be called with the ruleset file descriptor. If no error is returned, the calling thread is restricted with a new Landlock domain, which is composed of the parent domains and the provided ruleset, and its future children will inherit the same restrictions (see section 6.2). Updating the ruleset is still possible, but it will not have impact on any domain.

Listing 3: Enforce a Landlock ruleset on the calling thread

```c
if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
        error_exit("Failed to restrict privileges");

if (landlock_restrict_self(ruleset_fd, 0))
        error_exit("Failed to enforce ruleset");

close(ruleset_fd);
```

This sandbox scoping and the mandatory `NO_NEW_PRIVS` property enables us to get the innocuous property as defined in section 2.4.

7 Kernel implementation

As part of the Linux kernel, Landlock is implemented as an access control mechanism dedicated to create standalone sandboxes.

7.1 Linux Security Module

The Linux Security Module (LSM) framework provides a set of hooks for access control and kernel resource management. It also manages the

[4] Future work is planned to add a similar feature as *seccomp*'s `SECCOMP_FILTER_FLAG_TSYNC`: `https://github.com/landlock-lsm/linux/issues/2`

[5] As for *seccomp*, a thread can still call `landlock_restrict_self()` if it has `CAP_SYS_ADMIN`.

set of LSMs (e.g., SELinux, AppArmor) which are configured at build or at boot time. Especially, it makes it possible to run some of these LSMs together so that an access request can be processed by all currently stacked LSMs. Landlock is one of these stackable LSM, which means that it can be used with any other LSM. This is an important property because Landlock is a new layer of security. It is not meant to replace existing ones, and users should not choose between Landlock or another security mechanism. Developing this first (access control) stackable LSM was possible thanks to a community effort to improve the LSM framework [38].

7.2 Implicit restrictions

As explained in section 2.5, a sandboxing mechanism should not allow impersonation of processes outside of the sandbox. On Linux, the `ptrace()` system call can alter another process, leading to impersonation. Processes restricted by Landlock cannot request to trace processes not part of the same Landlock domain or a child one, which can only have more restrictions. For confidentiality and integrity reasons, accessing data or resources from processes outside the sandbox is also denied. These restrictions also apply to other kernel interfaces such as `/proc`: process's memory, file descriptors…

7.3 Filesystem access control

Ephemeral labeling A sandboxing mechanism needs to map access rights to a set of files, either to allow or to deny access. At the same time, a sandbox's lifetime is bound to the lifetime of the contained processes, so no trace (on the filesystem) of the related security policy must remain after that.

It is not possible to label files in a persistent way for several reasons:

— multiple concurrent policies can be enforced at the same time, from different applications or even different versions of the same application;

— new files are not owned by the application sandboxing itself but the user launching it;

— some files can be owned by other users, for instance system files owned by *root*;

— some files can only be accessible in a read-only way;

— some files are served by synthetic filesystems (e.g., `/proc`), which means that they are not backed by a storage device.

— some other files are served through the network (e.g., NFS), and then not fully controlled nor trusted by the local kernel.

Therefore, a sandboxing implementation cannot rely on file metadata such as regular file permissions, ACLs, or extended attributes (i.e. *xattr*) like used by SELinux and Smack.

Linux is a flexible kernel that empowers users to create namespaces for different kernel resources. The mount namespace creates a virtual filesystem topology exposed to a set of processes, for instance in a container. Therefore, processes may have different file topology or root directory. Moreover, namespaces of a process can change during its lifetime, which makes it impractical to create rules tied to namespaces. Furthermore, relying on file paths relative to the init namespace could expose to side-channel attacks against other access control systems (e.g., infer other restrictions because of Landlock). Because sandboxed processes can be in a mount namespace, a sandboxing security policy cannot be defined with absolute file paths like used by AppArmor and Tomoyo.

Considering the defined sandboxing constraints, a new way to identify files for Landlock was designed. Labeling must be ephemeral and only stored in memory to make it possible to tie each security policy to the lifetime of the related sandboxed processes. User space passes a file descriptor to the kernel through the `landlock_add_rule()` syscall. The kernel then looks at the related inode and ties it to either a new Landlock object or an existing one. This Landlock object is used as a generic kernel object identifier owned by all Landlock rules identifying this inode and then tied to their lifetimes. Because files can disappear (e.g., deleted or unmounted), Landlock objects are implemented as weak references to kernel objects, which makes this mechanism light and race condition free.

A ruleset is dynamic and is mainly implemented as a red-black tree. For now, a domain uses the same data type as a ruleset (to make it simpler), but a better approach would be to use a hash table because domains are immutable. The handled access rights of the inherited parent domains are stored in a flexible array member. A similar stack of access rights is used for rules, but this time to identify allowed accesses. These stacks keep a complete view of the composed rulesets, which are required both for correct policy composition and for auditing. Indeed, being able to tell which (parent) domain is denying an access request is very useful for application developers, kernel developers, system administrators, distribution maintainers, security experts, or power users. This property will be critical to better understand the reason of denials with the audit framework. [6]

Access rights of opened files When a sandboxed task opens a file, it gets a new file descriptor if the open mode (read or write), matches `LANDLOCK_ACCESS_FS_READ_FILE`, `LANDLOCK_ACCESS_FS_READ_DIR`, or `LANDLOCK_ACCESS_FS_WRITE_FILE`. Because this open mode cannot be changed for an opened file (`struct file`), it is guaranteed that the security policy will be enforced by every part of the kernel without additional changes. However, some operations on opened files might not be controlled by the open mode. For instance, most `ioctl` operations are not related to the read nor write semantics of regular files.

Moreover, because Landlock is incrementally gaining more access rights, some might initially not be handled (i.e. always allowed) and become controllable with a new version of the kernel. For instance, the `truncate` operation was initially allowed and because such an operation can be performed on a file path or a file descriptor, it was required to tie the new `LANDLOCK_ACCESS_FS_TRUNCATE` right to the opened file. This has implications for the access rights check because the open operation might be allowed but not the following truncate operation. Some optional access right must then be collected but not necessarily checked at open time.

As explained in section 2.5, less restricted processes need to be protected against confused-deputy attacks. For instance, a file opened by an unsandboxed process and then passed to a sandboxed process should not get any restrictions. Similarly, if a file is opened by a sandboxed process without the `LANDLOCK_ACCESS_FS_TRUNCATE` right, and then passed to an unsandboxed process, and then returned to the sandboxed process, the truncate operation must still not be allowed on this opened file. This means that it must never be allowed to truncate this opened file even in the unsandboxed process receiving it. Scoped restriction may not only apply to the sandbox but to the whole system that could share some resources with the sandbox.

Access rights tied to opened files are stored as a bitmask, which makes it efficient because of its small memory footprint and locality. This means that even unsandboxed tasks with limited access to such opened files will not pay a noticeable performance impact, which is in line with the principles defined in section 5.5.

[6] Audit support was part of the initial design and is currently actively being worked on.

Composition of file hierarchy restrictions Landlock domains have references to a set of rules that can identify inodes. When access is requested by sandboxed processes, all these restrictions must be considered. Starting with the leaf file or directory of a path, Landlock walks towards the real root directory, considering the different mount points of this path, but ignoring the covered ones (i.e. one mount point over another). For each of the walked inode, if a `struct landlock_object` is tied to it, then Landlock looks if the current domain has a reference to it. If this is the case, the related allowed access is looked at to see if it matches the request. If a domain's layer handles this access and one of its rules is found and matches for this inode, then this layer is marked as allowing the request. The path walk continues until all layers grant access to the request.

Let's say a session manager creates a first layer of sandboxing for logged users. In figure 4, this security policy must be quite permissive because users should be allowed to do anything with their files. However, a security policy with a list of allowed file hierarchies can still be enforced: common system directories in read-only, `/home` and temporary directories in read-write. This gives us some basic security guarantees: for instance, users would not be able to access files in /boot or test files forgotten by the system administrator at the root directory.

Then a user can launch an application with a sandbox manager (e.g., Firejail). In figure 5, according to a dedicated profile, the new application instance can initially only be allowed to access files potentially required: the application's libraries, the system's and user's configurations, a cache directory, and the directory used to store pictures. This application can now only access the specified files, but not sensitive ones (e.g., user's SSH private keys).

In figure 6, the newly launched application instance can finally sandbox itself to tailor its access according to user's provided arguments: the image to display. Now, only the cache and the specified picture can be accessed. If the picture includes an exploit to take control of the application (e.g., because of a bug in a parser), the malicious code would only be allowed to access the picture in a read-only mode, and the related cache in read-write mode.

In figure 7, the request to open the *cool.jpg* file must be approved by all three sandbox layers. This defense in depth approach helps mitigate the security impact of exploits thanks to complementary scoped accesses. For instance, if a first exploit takes control of the sandboxed application's process and can write a second exploit in the cache, then this attack could persist until the next launch. This could then be used to create a persistent attack that could potentially bypass the third sandboxing layer by not creating it in the first place, but it would not be able to bypass the parent layers.

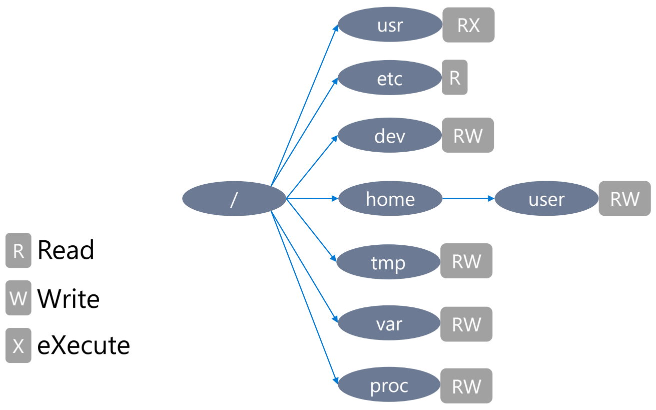


Fig. 4. Nested sandboxes: first layer

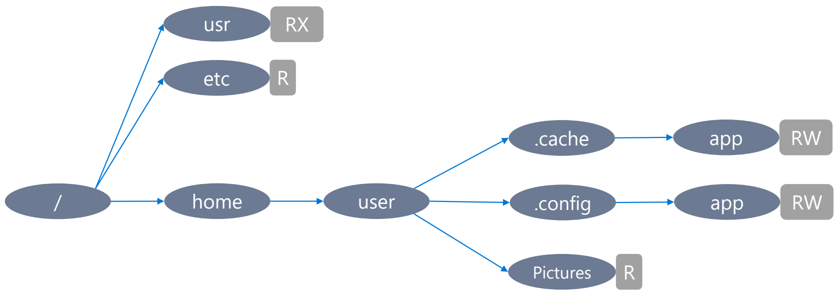


Fig. 5. Nested sandboxes: second layer

File topology changes Several nested sandbox layers can define different restrictions on the same file hierarchy. Because the rules that defined the related allowed accesses are tied to inodes, being able to change the parent of a file hierarchy from a less privileged rule to a more privileged rule would allow to change the related files in a way that was not allowed at the ruleset creation time. Reparenting a file hierarchy can be done with either a bind mount, a hard link, or a rename action.

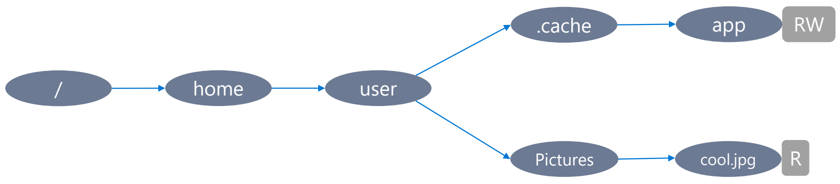


Fig. 6. Nested sandboxes: third layer

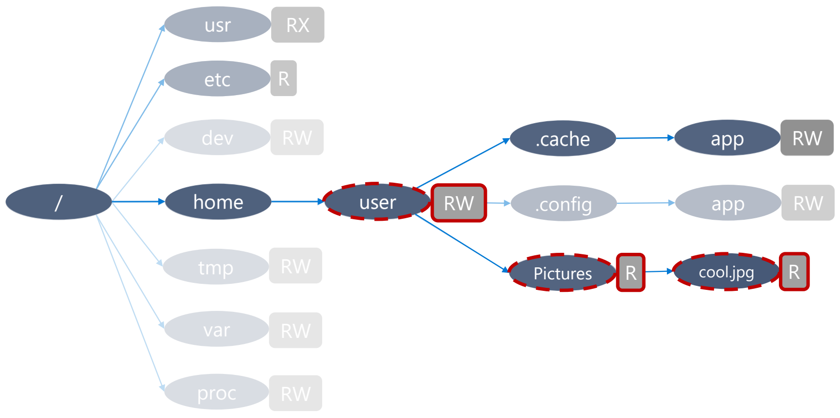


Fig. 7. Nested sandboxes: check all three layers

Bind mount exposes a file hierarchy several times as subsets of other file hierarchies. `pivot_root` swaps the root directory of the current mount namespace with an arbitrary directory. Both actions can change the file topology of a mount namespace, and they must then be restricted by Landlock to avoid policy bypasses. Any `mount` or `pivot_root` actions are currently denied for any sandbox.[7] However, just changing the root directory of a mount namespace does not put the Landlock security policies at risk because it only scopes the file topology to a subset of the original one. For this reason, `chroot` is not denied, even if it requires Linux privilege.

The `link()` system call makes a file visible in several directories at the same time. The `rename()` system call moves a file or a directory without copying it, which means to change its parent directory. Both features may change the file topology when the new parent directory changes (not just the file name). These actions are then allowed when there is no reparenting, but it was initially denied when linking or renaming files or directories otherwise.

[7] `https://github.com/landlock-lsm/linux/issues/14`

To allow file links and renames, the source and destination file hierarchy must have the `LANDLOCK_ACCESS_FS_REFER` right, but this is not enough. Indeed, moving or linking a file must not result in a privilege escalation because of new allowed access rights inherited from the new file hierarchy. Partial ordering for layers of file hierarchy accesses were implemented to be able to know if there are more privileges provided for a file hierarchy than another. This way, Landlock can allow a file to be linked or renamed if the destination file hierarchy would not give more access to this file.

Without Landlock, links and renames only make sense when the source and the destination are on the same mount point, not only the same filesystem. When such an action is requested on different mount points, the kernel denies the request with the `EXDEV` error code to inform user space about the reason. When user space gets this error code for a `rename` action, it should fall back to copying the source to the destination and removing the source file. Landlock leverages this compatibility mechanism by returning the `EXDEV` code when the source or destination does not have the `LANDLOCK_ACCESS_FS_REFER` right. However, if the fallback would not succeed because `LANDLOCK_ACCESS_FS_MAKE_*` (according to the file type) is missing for the destination, then Landlock returns the `EACCES` error code. Additionally, for a `rename` operation, `LANDLOCK_ACCESS_FS_REMOVE_FILE` must be allowed on the source.

7.4 Network access control

Starting with Linux 6.7, a Landlock ruleset can handle two new access rights: `LANDLOCK_ACCESS_NET_BIND_TCP` and `LANDLOCK_ACCESS_NET_CONNECT_TCP`. When handled, the related actions are denied unless explicitly allowed by a `struct landlock_net_port_attr` rule for a specific port.

This initial network support is simple and useful for most applications using TCP. Defining restrictions based on a set of ports is interesting because it identifies a set of well-defined services.[8] From an application developer point of view, most of the time it does not really make sense to restrict access to a specific peer, which might be defined with a name, resolved by a DNS client, but not the kernel.

Access rights are not tied to opened sockets but checked at `bind` or `connect` call time against the caller's Landlock domain. For the filesystem,

[8] It is of course possible for any service to use any available TCP port, but IETF's RFC 1340 exists for Internet services to be publicly reachable with a well-known ports.

an opened file is direct access to data. However, for network sockets, it cannot be identified for which data or peer a newly created socket will give access to. Indeed, only a connect or bind request makes it possible to identify the use case for this socket. Likewise, a directory file descriptor may enable us to open another file (i.e. a new data item), but this opening is also restricted by the caller's domain, not the opened directory's access rights.

When a domain contains only network restrictions (on all layers), then there is no restriction on file topology change (see section 7.3).

7.5 Complexity

Policy composition, and especially efficient nested sandboxing, is challenging. For instance, *seccomp* supports stacking but the stacked filters are executed one after the other, which leads to a $\mathcal{O}(n)$ complexity with n as the number of layers, and then a high performance impact.

Because Landlock's security policies are a set of simple rules (instead of complex BPF programs), n rulesets can be merged and create a new one containing all composed constraints. For the worth case scenario, lookup complexity is $\mathcal{O}(\log n)$ with n as the number of layers, which results in negligible performance impact.

In practice, the evaluation of a rule may require reading n access rights per kernel object, but because they are stored aligned (same locality) and take $16\ \textit{layers} \times 16\ \textit{potential rights} = 256\ \textit{bits} = 32\ \textit{bytes}$, it is really quick to read and compute compared to the other kernel operations (e.g., file path walk).

For simple rule types such as TCP ports, all layers could be merged to get $\mathcal{O}(1)$. However, this approach was not chosen for debugging and auditing reasons. Indeed, identifying which domain denies an action requires storing this mapping and getting it when access is requested.

The most complex part was the `LANDLOCK_ACCESS_FS_REFER` right [34]. Because of the way Landlock identifies files by their hierarchy, linking or renaming files in a way that would change their parent directories could make existing files available in directories where a different access policy applies. To avoid potential policy bypasses, Landlock needs to check that files do not gain additional access rights in their new locations. A partial ordering with potential nested layers was implemented to make sure that a destination would not inherit new access rights.

Composing sibling sandboxes might seem easy if the access control architecture is focused on specific subjects (i.e. processes), but resource passing needs to be considered. Passing file descriptors around processes should be allowed but not to gain new access exploiting a confused deputy vulnerability (see section 2.5). In the case of the `LANDLOCK_ACCESS_FS_TRUNCATE` right, potential access rights of opened files must be collected and saved even if there are not requested at open time because they could be required in a following system call with the same opened files. This provides an access control system consistent with the common read and write modes for opened files.

7.6 Testing and fuzzing

While developing Landlock, a lot of tests were implemented with the Kselftest framework. Test coverage for the Landlock code part of Linux 6.7 (released in 2024) is more than 92% of lines,[9] which is the maximum of what user space tests can cover. Indeed, the remaining lines are part of race condition checks, kernel runtime error checks (e.g., failed memory allocation), [10] or runtime safeguards required for guarantees not provided by the C language (e.g., all `WARN_ON_ONCE()` calls that should never be reachable). Comparing Single Lines of Code (SLOC) on Linux 6.7, there are around 2000 SLOC for the Landlock implementation (`security/landlock`) against 5400 SLOC for Landlock tests (`tools/testing/selftests/landlock`). This represents 210 test cases sharing more than 1500 assertions, which makes Landlock one of the best-tested subsystem with Kselftest.

Fuzzing the syscall interface is very important to find potential issues that could be used by attackers. However, fuzzing needs to be efficient by tweaking the inputs in a smart and relevant way. *syzkaller* is an unsupervised coverage-guided kernel fuzzer that was initially developed with the Linux kernel in mind. We contributed to *syzkaller* by defining the Landlock syscalls and writing some corner-case tests to improve coverage. Thanks to these changes, *syzkaller* coverage for the Landlock code part of Linux 6.7 is currently 71%. [11] Fuzzing helped find an issue while developing, and it gives some guarantees with all new kernel versions that the Landlock syscalls could not be used to attack the kernel.

7.7 Limitations

Design scope Landlock is part of the Linux kernel, which means that it can only control kernel resources. Consequently, user space services are out

[9] According to GCC/gcov version 13.

[10] Some of the runtime errors could be reached by user space, but this requires injecting faults into the kernel: `https://github.com/landlock-lsm/linux/issues/22`

[11] `https://syzkaller.appspot.com/upstream/manager/ci-qemu-upstream`

of scope: display server, sound server, DNS, user management… These services should be controlled with a dedicated user space access control system such as Polkit, complementary to Landlock.

Ongoing work For now, the main limitations of file access control are metadata access (e.g., file properties, extended attributes) and file path probing. However, Landlock can fully control access to file contents. We are working on extending Landlock's access control with new rights for already supported subsystems (i.e. file, TCP), but also to support new subsystems (e.g., task signaling, sockets, IPCs). [12]

Performance and usability improvements are also planned, especially with path walk access caching, tailored data types, and audit support to improve debuggability and provide metrics [37].

8 Upstreaming

Upstream refers to maintainers of software, whereas downstream refers to consumers of this software. Bringing changes to an open-source software should include the process of upstreaming as early as possible.

8.1 Why integrate changes in Linux mainline?

From a pragmatic point of view, there are at least three reasons to push changes to the mainline project:

- to make them available to all project's users,
- to get help and reviews improving quality,
- to limit maintenance cost.

Of course, contributing back to something we use is also great motivation and it gives visibility.

Merging our changes in the main project makes them available to the whole project community, including all downsteam users. In the case of Landlock, this is our goal: to protect as many users as possible.

8.2 Development workflow

A huge project Any software, open-source or not, may have a set of principles and rules. This may include coding style, code of conduct, version control usage, documentation, tests and more [15]. The Linux kernel is the largest open-source project in the world. This requires appropriate

[12] `https://github.com/landlock-lsm/linux/issues`

management. For instance, during the 10 weeks of release preparation, contributions from around 2000 developers may be merged, which may result in the addition of more than 500000 lines of code with more than 17000 commits [11]. The kernel source contains several subsystems, each of them managed by a set of maintainers. Subsystems can also be part of other subsystems, e.g., the network subsystem includes the eBPF subsystem. Even if there are multiple efforts to get a global consistency across all Linux kernels, in practice the subsystems rules may vary. As a result, contributing to different subsystems can sometime lead to inconsistent requirements.

The security subsystem The Linux Security Module framework is mainly an API shared by several security subsystems (e.g., Linux capabilities, SELinux, AppArmor, Landlock). It defines security hooks, enforcement points, and shared data types. It is maintained with the security subsystem and evolves over time according to the requirement of its users. Adding a new access control system may imply adapting this framework, which means changing code in other subsystems (e.g., filesystem, network). Indeed, security cannot be isolated to a set of files or a part of the kernel.

Emails, Git, and tooling Unlike most open source development, kernel discussions and code reviews mainly happen on mailing lists. A patch series is sent by email for each consistent set of changes, creating an email thread. From this thread, reviewers can start a discussion in a free format.

Being able to scale with a huge volume of contributions is challenging and emails happen to work fine [16]. A lot of kernel maintainers are very efficient at querying and filtering emails, which enables them to lead wide communities. Emails are also flexible and inclusive in the sense that it makes it easy to add new people or mailing lists to an existing discussion or code review, without requiring an account on a specific platform tied to specific rules. This decentralized architecture helps improve reliability of development and makes archiving easy.[13]

However, while emails are the initial medium, Git repositories are required for maintainers to send pull requests with signed tags to the maintainer of the parent subsystem (e.g., Linus Torvalds at the top). Git was initially created for Linux development and has since been adopted by most developers. Pull requests must have been tested in the *linux-next*

[13] There are ongoing discussions about alternatives to emails, but change comes with a cost proportional to the size of the project and the number of people and organizations involved.

repository, and it is the responsibility of the maintainer requesting the merge to check that everything work as expected. Of course, full testing is done for each new release.

A set of online tools help navigate with flows of contributions (e.g., *lore*, *patchwork*), public services run tests (e.g., LKP/0-Day CI, KernelCI, *syzbot*), and developer tools help working with patches (e.g., *Git*, *b4*, *lei*, email client with custom scripts).

8.3 How to contribute?

Conference talks and articles are a great way to understand the current state of development of a specific subsystem. The user space API documentation[14] (including man pages), user space code (tests, samples), and libraries documentation are useful to understand the design of the interface (e.g., syscall, synthetic filesystem) which is key for usability and maintainability of a subsystem.

To actively contribute, it is important to understand the development process [15], and especially how the subsystem's community works. Reading mailing list discussions related to merged features is a good start, and this can be eased with the `Link` tags in commit messages. Identifying maintainers and active contributors can help follow the project's direction. Some subsystems have a bug tracker that can ease this process.[15] This can be used to identify areas for first contributions (*good first issue*) and start with small but useful patches (e.g., fix issues, improve code or user documentation). Such contributions may target another part of the kernel from which the target subsystem would benefit, with others.

To minimize the cost of review and maintenance, good quality contributions must include tests, documentation, and helpful commit descriptions in consistent and bisectable patches. However, before investing too much in a complex feature and the related tests and documentation, it might be a good idea to start with a Request For Comments (RFC).

Private discussions with maintainers or contributors can help first contributions, but keep in mind that most discussions, especially reviews, should take place in public, which means on a related mailing list for Linux. Contributions may take time to land but keep going, take feedback from reviews and comments to learn, improve, and build reputation, it's worth it!

[14] `https://docs.kernel.org/userspace-api/landlock.html`
[15] `https://github.com/landlock-lsm/linux/issues`

8.4 Initial review cycles and design evolutions

Landlock development started in 2016 as an extension of *seccomp* [25]. This initial approach was to improve an existing mechanism with new features, but we quickly found out that this was not a good approach in the long run. The main issue is that the syscall layer is not a good place for checks related to kernel semantics.

With the second version of the patch series, we switched to the LSM framework, eBPF and *cgroups*. This new direction was promising thanks to the powerful and flexible eBPF engine [26].

In 2018, the eighth version of the patch series added support for file path identification with dedicated eBPF helpers. After that, it was mostly patch shrinking to reach a Minimum Viable Product (MVP).

In 2020, we revamped the whole patch series without eBPF, adding a new dedicated system call. eBPF is very powerful and can be leveraged by attackers against the kernel (e.g., verifier bugs, Spectre), which makes it unfit for unprivileged users [8]. Programmable interface with I/O (e.g., eBPF maps) can lead to side channel attacks against other programs (see section 5.5). Moreover, it is not possible to efficiently compose programs at run time but only to stack them (cf. *seccomp*), which would make eBPF use for sandboxing inefficient (see section 7.5). Anyway, this work on eBPF was still useful for the next versions of the patch series, and it contributed to bootstrap the BPF LSM, previously called Kernel Runtime Security Instrumentation (KRSI [39]).

With the 21st patch series, we switched to 3 dedicated syscalls (see section 6.5) to avoid a syscall multiplexer, and we improved the user space interface. Finally, the 34th patch series was merged [9, 40] and released with Linux 5.13 in 2021.

8.5 Maintenance and contributions

Upstreaming a major change like Landlock to the Linux kernel is only one of the first steps to make it widely available. Becoming maintainer implies a responsibility to lead the development of a part of Linux, to fix issues, to add documentation, to improve quality, and to mentor contributors. To educate kernel maintainers and Linux users about Landlock, we gave conference talks and workshops [26–32, 34–37]. To make kernel development and testing easier, especially for newcomers, we created and shared a set of standalone tools.[16] Because following development on

[16] `https://github.com/landlock-lsm/landlock-test-tools`

mailing lists might be a daunting task, we are maintaining a set of tasks on GitHub.[17] We also write newsletters to track updates, and we maintain a set of libraries to make it easier and safer for users to use Landlock.

As explained in section 6.3, the first version of Landlock was an MVP. We are now working on new features to improve the state of sandboxing on Linux and protect as many users as possible.

9 Adoption

Making Landlock broadly available on Linux systems is a prerequisite to protect as many users as possible.

9.1 Linux distributions and container runtimes

Because Landlock is an LSM, it can be enabled or disabled at boot time with a kernel command line parameter. Even if there is a default command line parameter in the mainline kernel, it is often overloaded by Linux distributions. The two steps to enable it in a Linux distribution were then to enable it in the kernel build configuration and the kernel boot configuration. The build configuration was the easy part to ask, but the default boot command line was a bit more challenging.

Landlock is currently available with the most generic Linux distributions: Ubuntu 22.04 LTS, Fedora 35, Arch Linux, Alpine Linux, Gentoo, Debian Sid, chromeOS, Azure Linux (CBL-Mariner), and WSL2.

Another unexpected challenge was to make the Landlock system calls available to processes running in container runtimes. Indeed, most of them now filter syscalls with *seccomp*. It was then required to propose the three new Landlock syscalls to be allowed. Landlock is currently supported within the containers of the following runtimes: Docker (Moby), Podman, *runc* (Open Container Initiative), LXC, and Incus.

9.2 Development tools, libraries, and documentation

As a new kernel feature, it was required to update some developer tools such as *strace* which requires up to date syscall signatures and other kernel interfaces' information.

To make Landlock easier to use, we developed one library for Rust [35][18] and another for Go [20].[19] The community has been actively developing support for other languages: Haskell, Python, and C.

[17] `https://github.com/landlock-lsm/linux/issues`

[18] `https://github.com/landlock-lsm/rust-landlock`

[19] `https://github.com/landlock-lsm/go-landlock`

Finally, for a new kernel feature to be developer-friendly, good documentation is required. The main Linux documentation is stored and maintained along the kernel source code [33]. However, the *man pages* are a separate project with a different documentation file format, and we also need to populate the Landlock-related pages and keep them up to date.

9.3 Sandboxed software

Landlock is still a new kernel security feature but there are already early adopters leveraging it. As for any open-source component, it is not easy to identify users, but it is still possible to get some clues for open-source communities.

Open-source and public-facing products include chromeOS, Nomad, and Polkadot. Microsoft is also using Landlock to protect Azure. Excluding developer tools and libraries (see section 9.2), a few open-source software programs support Landlock, such as Suricata [17], *sslh*, and XZ Utils. There is also an ongoing effort to add Landlock support to JavaScript, TypeScript, and WebAssembly runtimes to sandbox their execution [1–3]. Another initiative is to bring Landlock's capabilities to the Open Container Initiative's specification and runtime (*runc*), and PAM. Finally, most use cases may use Landlock with sandbox managers such as Minijail or Firejail, which boosts adoption and use cases.

Even if Landlock is designed for security, its standalone features can give rise to unexpected use cases such as hermetic compilations with *landlock-make* [41].

This list is probably the tip of the iceberg, and we will need to wait for more widely available sandbox tools to get a better idea of Landlock's users. Moreover, critical features were needed, and some important ones would help for wider adoption: file reparenting for efficient filesystem use (supported by ABI v2), file truncation to protect against file's content erasing (supported by ABI v3), audit support to more easily debug and report denials,[20] and an audit-only mode to get guarantees that a workflow or a fleet will work as expected before enforcing restrictions.[21]

9.4 The XZ backdoor

XZ Utils is a widely used compression tool and library. In March 2024, a backdoor was found and reported. It was introduced in February by a new maintainer who earned this trust after more than two years of effort.

[20] `https://github.com/landlock-lsm/linux/issues/3`
[21] `https://github.com/landlock-lsm/linux/issues/17`

Among the malicious changes, the attacker disabled Landlock's support for XZ Utils [12]. The sabotaged configuration check has since been fixed,[22] but this effort to stealthily disable sandboxing is a clear sign that Landlock disturbs attackers.

10 Conclusion and future work

As the Linux sandboxing feature, Landlock can help protect users against security vulnerabilities or malicious applications. It is designed to fit well with embedded sandboxing but it can also create several layers of security, following the defense in depth principle.

Landlock empowers the Linux community to protect itself with defensive tools. To speed up the process of sandboxing applications, one of the next steps is to create an easy-to-use sandboxer with flexible security policies.

Landlock is already an efficient sandboxing mechanism with a lot of potential. We are working on new features to improve it and new contributors are welcome!

Acknowledgments

I would like to thank Günther, Nicolas, Praveen, and Wei for their reviews.

References

1. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Hardening WASI using Landlock LSM. In *USENIX Security Poster Session*, 2022. `https://cs.unibg.it/seclab-papers/2022/USENIX/wasi-poster.pdf`.
2. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2023. `https://doi.org/10.1145/3579856.3595799`.
3. Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. NatiSand: Native Code Sandboxing for JavaScript Runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. Association for Computing Machinery, 2023. `https://doi.org/10.1145/3607199.3607233`.

[22] `https://github.com/tukaani-project/xz/commit/f9cf4c05edd1`

4. Dionysus Blazakis. The Apple Sandbox. In *Black Hat DC*, 2011. `https://media.blackhat.com/bh-dc-11/Blazakis/BlackHat_DC_2011_Blazakis_Apple_Sandbox-wp.pdf`.
5. Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *11th USENIX Security Symposium*, 2002. `https://www.usenix.org/conference/11th-usenix-security-symposium/setuid-demystified`.
6. Jonathan Corbet. Controlling access to user namespaces, 2016. `https://lwn.net/Articles/673597/`.
7. Jonathan Corbet. The inherent fragility of seccomp(), 2017. `https://lwn.net/Articles/738694/`.
8. Jonathan Corbet. Reconsidering unprivileged BPF, 2019. `https://lwn.net/Articles/796328/`.
9. Jonathan Corbet. Landlock (finally) sets sail, 2021. `https://lwn.net/Articles/859908/`.
10. Jonathan Corbet. A security-module hook for user-namespace creation, 2022. `https://lwn.net/Articles/903580/`.
11. Jonathan Corbet. Some 6.7 development statistics, 2024. `https://lwn.net/Articles/956765/`.
12. Russ Cox. Timeline of the xz open source attack, 2024. `https://research.swtch.com/xz-timeline`.
13. FreeBSD. libcasper. `https://man.freebsd.org/cgi/man.cgi?query=libcasper&sektion=3`.
14. Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS Symposium*, 2003. `https://www.ndss-symposium.org/ndss2003/traps-and-pitfalls-practical-problems-system-call-interposition-based-security-tools/`.
15. The kernel development community. Working with the kernel development community, 2024. `https://docs.kernel.org/process/`.
16. Greg Kroah-Hartman. Patches carved into stone tablets. In *Kernel Recipes*, 2016. `https://kernel-recipes.org/en/2016/talks/patches-carved-into-stone-tablets/`.
17. Eric Leblond. Attaques de type Supply Chain sur Suricata. In *SSTIC*, 2023. `https://www.sstic.org/2023/presentation/attaque_supply_chain_suricata/`.
18. Microsoft. AppContainer isolation, 2023. `https://learn.microsoft.com/en-us/windows/win32/secauthz/appcontainer-isolation`.
19. Microsoft. Microsoft Defender Application Guard overview, 2023. `https://learn.microsoft.com/en-us/windows/security/application-security/application-isolation/microsoft-defender-application-guard/md-app-guard-overview`.
20. Günther Noack. Why use Go-Landlock for sandboxing? In *Zurich Gophers Meetup*, 2022. `https://blog.gnoack.org/post/go-landlock-talk/`.
21. Günther Noack. Landlock: Best Effort mode, 2023. `https://blog.gnoack.org/post/landlock-best-effort/`.
22. Committee on National Security Systems (CNSS). CNSS Glossary, 2022. `https://www.cnss.gov/CNSS/issuances/Instructions.cfm`.

23. OpenBSD. pledge - restrict system operations. `https://man.openbsd.org/pledge.2`.
24. OpenBSD. unveil - unveil parts of a restricted filesystem view. `https://man.openbsd.org/unveil.2`.
25. Mickaël Salaün. [RFC v1 00/17] seccomp-object: From attack surface reduction to sandboxing, 2016. `https://lore.kernel.org/r/1458784008-16277-1-git-send-email-mic@digikod.net`.
26. Mickaël Salaün. Landlock : cloisonnement programmable non privilégié. In *SSTIC*, 2017. `https://www.sstic.org/2017/presentation/landlock/`.
27. Mickaël Salaün. Landlock LSM: Toward Unprivileged Sandboxing. In *Linux Security Summit North America*, 2017. `https://sched.co/BKOm`.
28. Mickaël Salaün. File access-control per container with Landlock. In *FOSDEM*, 2018. `https://archive.fosdem.org/2018/schedule/event/containers_landlock/`.
29. Mickaël Salaün. How to Safely Restrict Access to Files in a Programmatic Way with Landlock? In *Linux Security Summit North America*, 2018. `https://lssna18.sched.com/event/FLYR`.
30. Mickaël Salaün. Internals of Landlock: a new kind of Linux Security Module leveraging eBPF. In *Pass the Salt*, 2018. `https://2018.pass-the-salt.org/programme/#landlock`.
31. Mickaël Salaün. Deep Dive into Landlock Internals. In *Linux Security Summit*, 2021. `https://sched.co/11MXq`.
32. Mickaël Salaün. Sandboxing Applications with Landlock. In *Open Source Summit*, 2021. `https://osselc21.sched.com/event/lAVl`.
33. Mickaël Salaün. Landlock user space documentation, 2022. `https://docs.kernel.org/userspace-api/landlock.html`.
34. Mickaël Salaün. Update on Landlock: Lifting the File Reparenting Limits and Supporting Network Rules. In *Linux Security Summit North America*, 2022. `https://sched.co/11MXq`.
35. Mickaël Salaün. Backward and forward compatibility for security features. In *FOSDEM*, 2023. `https://fosdem.org/2023/schedule/event/rust_backward_and_forward_compatibility_for_security_features/`.
36. Mickaël Salaün. Landlock Workshop: Sandboxing Application for Fun and Protection. In *Linux Security Summit Europe*, 2023. `https://sched.co/1OLAi`.
37. Mickaël Salaün. Update on Landlock: Audit, Debugging and Metrics. In *Kernel Recipes*, 2023. `https://kernel-recipes.org/en/2023/update-on-landlock-audit-debugging-and-metrics/`.
38. Casey Schaufler and John Johansen. Namespacing and Stacking the LSM. In *Linux Plumbers Conference*, 2017. `https://blog.linuxplumbersconf.org/2017/ocw/sessions/4768.html`.
39. KP Singh. Kernel Runtime Security Instrumentation. In *Linux Security Summit Europe*, 2019. `https://sched.co/Tyn7`.
40. Linus Torvalds. Pull Landlock LSM, 2021. `https://git.kernel.org/torvalds/c/17ae69aba89dbfa2139b7f8024b757ab3cc42f59`.
41. Justine Tunney. Using Landlock to Sandbox GNU Make, 2022. `https://justine.lol/make/`.

42. Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, 2003. `https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/watson/watson.pdf`.

43. Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010. `http://www.trustedbsd.org/2010usenix-security-capsicum-website.pdf`.