Microsoft

Backward and forward compatibility for security features (illustrated with Landlock)

FOSDEM – Rust devroom

Mickaël Salaün

2023-02-04

Why care about security?

An innocuous and trusted process can become **malicious** during its **lifetime** because of bugs exploited by attackers. Every running app/service increases (user) **attack surface**.

Problem (as developers):

- We don’t want to participate to malicious actions through our software.
- **Responsibility** for users, especially to protect their (personal) data (e.g., pictures, ~/.ssh).
- What about potentially malicious **third-party** libraries?

Security sandboxing

A security approach to **isolate** a software component from the rest of the system by **dropping ambient access rights** which are not needed.

NB:

- Namespaces/containers are not considered security sandboxes per se, but tools to “virtualize” resources, with their own vulnerabilities.
- seccomp is not an access control system (e.g., no file access restrictions).

Landlock: the Linux sandboxing solution

Landlock is an access control system available to **unprivileged** processes on Linux, which enables developers to add **built-in** application **sandboxing**.

Create **new security layers** in addition to the existing system-wide access-control.

Available in mainline since 2021 (Linux 5.13), and enabled by default on multiple distros: Ubuntu 22.04 LTS, Fedora 35, Arch Linux, Alpine Linux, Gentoo, Debian Sid, chromeOS, CBL-Mariner, WSL2

Tailored and embedded security policy

Developers are in the best position to reason about the required accesses according to legitimate behaviors:

- Application semantics
- Static and dynamic configuration
- User interaction

Testable and can be kept in sync with evolving business logic **over time**.

The Rust library

Idiomatic Rust API leveraging strong typing and common patterns (e.g., builder for objects and their references).

Still working on getting the API right, especially with the compatibility constraints explained in this talk.

Some early public users of rust-landlock:

- Keysas: USB virus cleaning station
- Birdcage: Cross-platform embeddable sandboxing
- rust-wasm-landlock: An integration between Wasmtime and Landlock

Code examples

The following simple examples are correct but:

- The C code doesn’t check for compatibility issues whereas the Rust code does
- The Rust code doesn’t leverage the implicit access right conversion from a specific (and tested) Landlock version

Underneath, they rely on 3 new dedicated syscalls.

Step 1: Create a ruleset

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

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

```rust
Ruleset::new()
    .handle_access(make_bitflags!(
        AccessFs::{Execute | WriteFile}))?
    .create()?
```

Step 2: Add rules

```c
int err;
struct landlock_path_beneath_attr path_beneath = {
    .allowed_access = LANDLOCK_ACCESS_FS_EXECUTE,
};

path_beneath.parent_fd = open("/usr",
                              O_PATH | O_CLOEXEC);
if (path_beneath.parent_fd < 0)
    error_exit("Failed to open file");

err = landlock_add_rule(ruleset_fd,
        LANDLOCK_RULE_PATH_BENEATH, &path_beneath, 0);
close(path_beneath.parent_fd);
if (err)
    error_exit("Failed to update ruleset");
```

```rust
Ruleset::new()
    .handle_access(make_bitflags!(
        AccessFs::{Execute | WriteFile}))?
    .create()?
    .add_rule(
        PathBeneath::new(PathFd::new("/usr")?)
        .allow_access(AccessFs::Execute)
    )?
```

Step 3: Enforce the ruleset

```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);
```

```rust
Ruleset::new()
    .handle_access(make_bitflags!(
        AccessFs::{Execute | WriteFile}))?
    .create()?
    .add_rule(
        PathBeneath::new(PathFd::new("/")?)
        .allow_access(AccessFs::Execute)
    )?
    .restrict_self()?
```

Incremental development

Because it is complex, a new kernel access control system cannot implement everything at once.

Landlock is useful as-is and it is gaining new features over time, which may enable to either add or remove restrictions.

Restrictions evolution over versions

Always denied

- Get new privileges
- Ptrace a parent sandbox
- Change FS topology
- Reparent files

Configurable

- Read file
- Write file
- …

Always allowed

- Change directory
- Read file metadata
- Change file ownership
- IOCTL
- Truncate file
- …

Landlock v1

Restrictions evolution over versions

Always denied

- Get new privileges
- Ptrace a parent sandbox
- Change FS topology
- ~~Reparent files~~

Configurable

- Read file
- Write file
- …
- Reparent files

Always allowed

- Change directory
- Read file metadata
- Change file ownership
- IOCTL
- Truncate file
- …

Landlock v1

Landlock v2

Restrictions evolution over versions

Always denied

- Get new privileges
- Ptrace a parent sandbox
- Change FS topology
- ~~Reparent files~~

Configurable

- Read file
- Write file
- …
- Reparent files
- Truncate file

Always allowed

- Change directory
- Read file metadata
- Change file ownership
- IOCTL
- ~~Truncate file~~
- …

Landlock v1

Landlock v2

Landlock v3

Application compatibility

Forward compatibility for applications is handled by the kernel development process (quite like the Rust *#[non_exhaustive]* type attribute).

Backward compatibility for applications is the responsibility of their developers, who may not be aware of the **kernel on which their application will run**.

Each new Landlock feature increments the Landlock ABI version, which is useful to implement a fallback mechanism: **best-effort** approach.

Check the Landlock ABI version (in C)

```c
int abi = landlock_create_ruleset(NULL, 0, LANDLOCK_CREATE_RULESET_VERSION);

if (abi < 0)
    return 0;
```

Property #1: Ease of use

Defining the right access scope may be challenging and the API should help developers as much as possible.

Requirements:

- **Generic API** (and types) to incrementally **build** a set of access rules before enforcing them
- No knowledge of Landlock internals required: should **focus on required accesses**
- Enable **fine-grained and coarse-grained** access rights
- Simpler to write for common use cases

Group access rights per ABI

```
let abi = ABI::V2;

Ruleset::new()
    .handle_access(AccessFs::from_all(abi))?
    .create()?
    .add_rule(
        PathBeneath::new(PathFd::new("/usr")?)
        .allow_access(AccessFs::from_read(abi))
    )
```

Property #2: Strict restrictions

Detect and error out for any incompatibility.

Use cases:

1. For developers and CI tests, to be sure that sandboxing is not an issue for legitimate use
2. For security software, to be sure that a set of security properties are guarantee

Requirement:

- Being able to **force the whole sandboxing** or error out if a required feature is not supported (e.g., the refer access right for file reparenting).

Property #3: Best-effort with minimal requirement

Enforce restrictions as much as possible according to the running kernel: *don't break my application!*

Use case:

- For end users, **opportunistically sandbox** applications without error

Requirements:

- Being able to **disable the whole sandboxing** if a required feature is not supported (e.g., the refer access right for file reparenting).
- Make this approach easier to write

Property #4: Runtime configuration with maximum execution

Should be simple to set or unset at run time according to:

- Test environment (e.g., build profile, variables)
- User configuration

Help identify sandboxing specific code issues.

Requirement:

- Run the same code as much as possible (i.e., same behavior: check same files, make same syscalls…) but only enforce restrictions when requested.

1st approach: set_best_effort()

```
Ruleset::new()
    .handle_access(AccessFs::from_all(ABI::V1))?
    .set_best_effort(false)
    .handle_access(AccessFs::Refer)?
    .set_best_effort(true)
```

Pros:

- Flexible
- Easy to understand

Cons:

- Return an error right away instead of ignoring the whole sandbox when specific features are missing
- Change the ruleset behavior over build steps, which makes it likely to forget to “reset” it

2nd approach: set_compatibility()

```
Ruleset::new()
    .handle_access(AccessFs::from_all(ABI::V1))?
    .set_compatibility(CompatLevel::SoftRequirement)
    .handle_access(AccessFs::Refer)?
    .set_compatibility(CompatLevel::BestEffort)
```

Pros:

- Flexible
- Handle soft requirement: disable the whole sandbox when specific features are missing

Cons:

- More complex with three choices: *BestEffort*, *SoftRequirement*, *HardRequirement*
- Change the ruleset behavior over build steps, which makes it likely to forget to “reset” it

3rd approach: Ruleset constructor and Access attribute

```
Ruleset::new(CompatMode::ErrorIfUnmet)
    .handle_access(AccessFs::from_all(ABI::V1))?
    .handle_access(AccessFs::Refer.disable_sandbox_if_unmet(true))?
```

Pros:

- Flexible
- No need to reset the ruleset compatibility state
- Simpler and more explicit

Cons:

- A bit verbose when used

Going forward

- Improve ease of use with type inference
- Add new helpers
- Improve documentation
- **Sandbox your applications** and others'
  - Secure Open Source Rewards
  - Google Patch Rewards

Questions or ideas?

Current documentation: https://landlock.io/rust-landlock

Ongoing compatibility PR: https://github.com/landlock-lsm/rust-landlock/pull/12

Go talk: https://blog.gnoack.org/post/go-landlock-talk/

Past talks: https://landlock.io

landlock@lists.linux.dev

Thank you!