Microsoft

Azure

Sandboxing Applications with Landlock

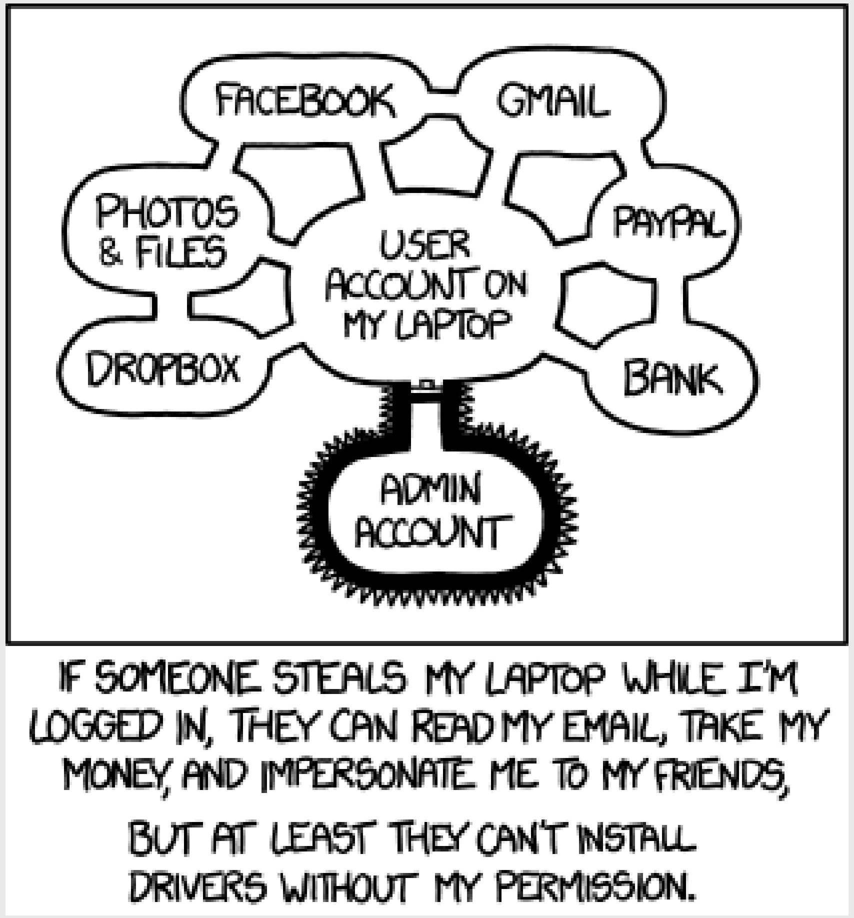
Open Source Summit

Mickaël Salaün

Senior Software Engineer

2021-09-28

User data


FACEBOOK
GMAIL
PHOTOS & FILES
PAYPAL
USER ACCOUNT ON MY LAPTOP
DROPBOX
BANK
ADMIN ACCOUNT
IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS,
BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.

https://xkcd.com/1200

What is it about?

This talk is not about kernel protection (e.g., bug fixing, integrity, hardening, attack surface reduction).

This talk is about access-control and risk mitigation for user space developers to protect their users: limit the damage of exploited bugs, which may be known or not, fixed by an ongoing update or not.

Plan

What are we trying to solve?

Why Landlock and what is it?

How to use Landlock?

Backward and forward compatibility

What is coming next?

State of security for applications nowadays

Initial assumptions: Trusted Computing Base

Applications chosen by users or sysadmins are **trusted**:

- Trust means that we trust the honesty and well-intention of the developers.
- There is multiple and **different levels of trust** and different consequences in case of a breach:
  - system data,
  - user data,
  - app data.
- It doesn’t mean that applications are bulletproofs nor safe to use in any conditions.

Supply chains are secure, the configuration is secure (enough), and other critical parts of the system (e.g., kernel, important services) are trusted and uncompromised.

State of security for applications nowadays

Threats we are tackling

An innocuous and trusted process can become malicious during its lifetime because of bugs, exploited by attackers or just triggered by users.

Application can be misused by (not malicious) users.

What is (security) sandboxing?

A security approach to isolate a software component from the rest of the system.

Threat models:

- Protecting from vulnerable code maintained by the developer.
- Protecting from malicious **third-party** code.
- Can be defined by the developers as it fit best.

Why do we need sandboxing?

Problem

Perfect security doesn’t fit with pragmatic development: bulletproof (and useful) software is costly and very difficult to achieve (if ever possible).

We don’t want to participate to malicious actions through our software, which might come from an exploited bug, a vulnerability, leveraged by an attacker.

Responsibility of user data

Helps protect what application developers oversee.

Why do we need sandboxing?

Solutions (complementary ones)

Reactive:

- Fix bugs quickly and push updates.

Proactive:

- Look for bugs (e.g., audit, fuzzing) and fix them.
- Add more tests and use them.
- Use safer languages, libraries and compilers.
- Consider (most) software as potentially malicious and **protect the rest of the system** from them.

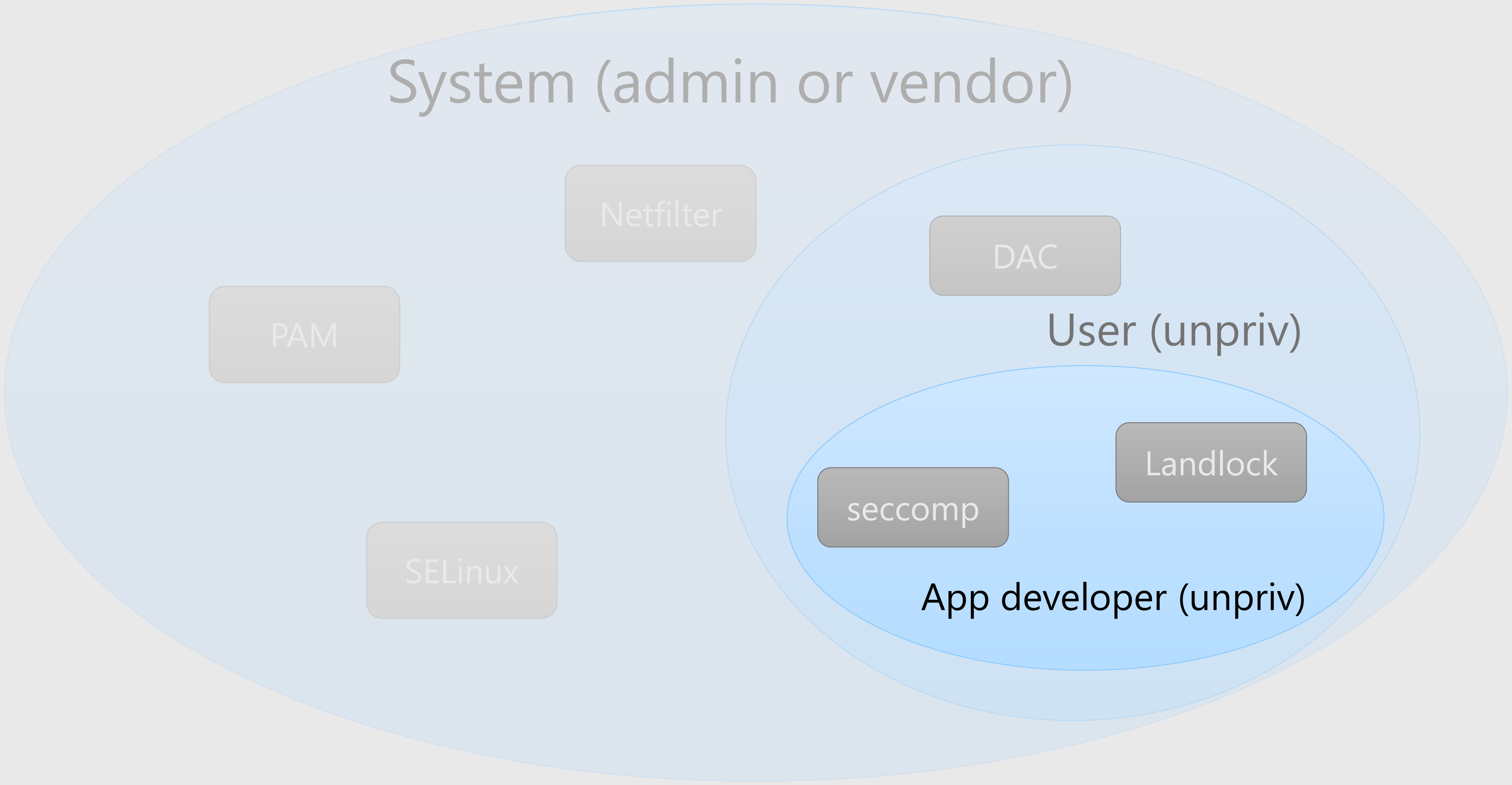
State of the art

Non-Linux systems (partial list):

- XNU Sandbox
- Windows's (Less Privileged) AppContainer
- FreeBSD's Capsicum
- OpenBSD's Pledge and Unveil

What about the Linux world?

Security features available in traditional Linux systems



Comparisons of different sandboxing mechanisms

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

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

What is Landlock?

A set of features (available since Linux 5.13) to create sandboxes:

Restrict ambient rights according to the **kernel semantic** (e.g., global filesystem access) for a set of processes, because seccomp is not enough.

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

Compose access-controls from multiple tenants (e.g., sysadmin, app developer, cloud clients).

Use cases

Interesting for **build-in** application sandboxing and **sandbox managers**.

Current

- Parsers hardening (e.g., archive tools, file format conversion, renderers)
- (Part of) applications with limited file renaming or linking needs (e.g., some system or network services)
- Monolith applications dealing with different levels of confidentiality (e.g., web browser, email server).

Next

- System services
- Generic container or sandbox managers (e.g., Docker, Flatpak, Firejail)

Current access-control features: filesystem

Allow a thread (and its future children) to access to a set of file hierarchies:

- Execute, read or write to a file
- List a directory or remove files
- Create files

Current limitations

- File reparenting: renaming or linking a file to a different parent directory is always denied
- Filesystem topology modification: arbitrary mounts

```
LANDLOCK_ACCESS_FS_EXECUTE
LANDLOCK_ACCESS_FS_WRITE_FILE
LANDLOCK_ACCESS_FS_READ_FILE
LANDLOCK_ACCESS_FS_READ_DIR
LANDLOCK_ACCESS_FS_REMOVE_DIR
LANDLOCK_ACCESS_FS_REMOVE_FILE
LANDLOCK_ACCESS_FS_MAKE_CHAR
LANDLOCK_ACCESS_FS_MAKE_DIR
LANDLOCK_ACCESS_FS_MAKE_REG
LANDLOCK_ACCESS_FS_MAKE_SOCK
LANDLOCK_ACCESS_FS_MAKE_FIFO
LANDLOCK_ACCESS_FS_MAKE_BLOCK
LANDLOCK_ACCESS_FS_MAKE_SYM
```

Automatic hierarchy restrictions

Multiple sandbox layers

All applications (e.g., shells) are allowed to create their own sandbox, which may create hierarchies.

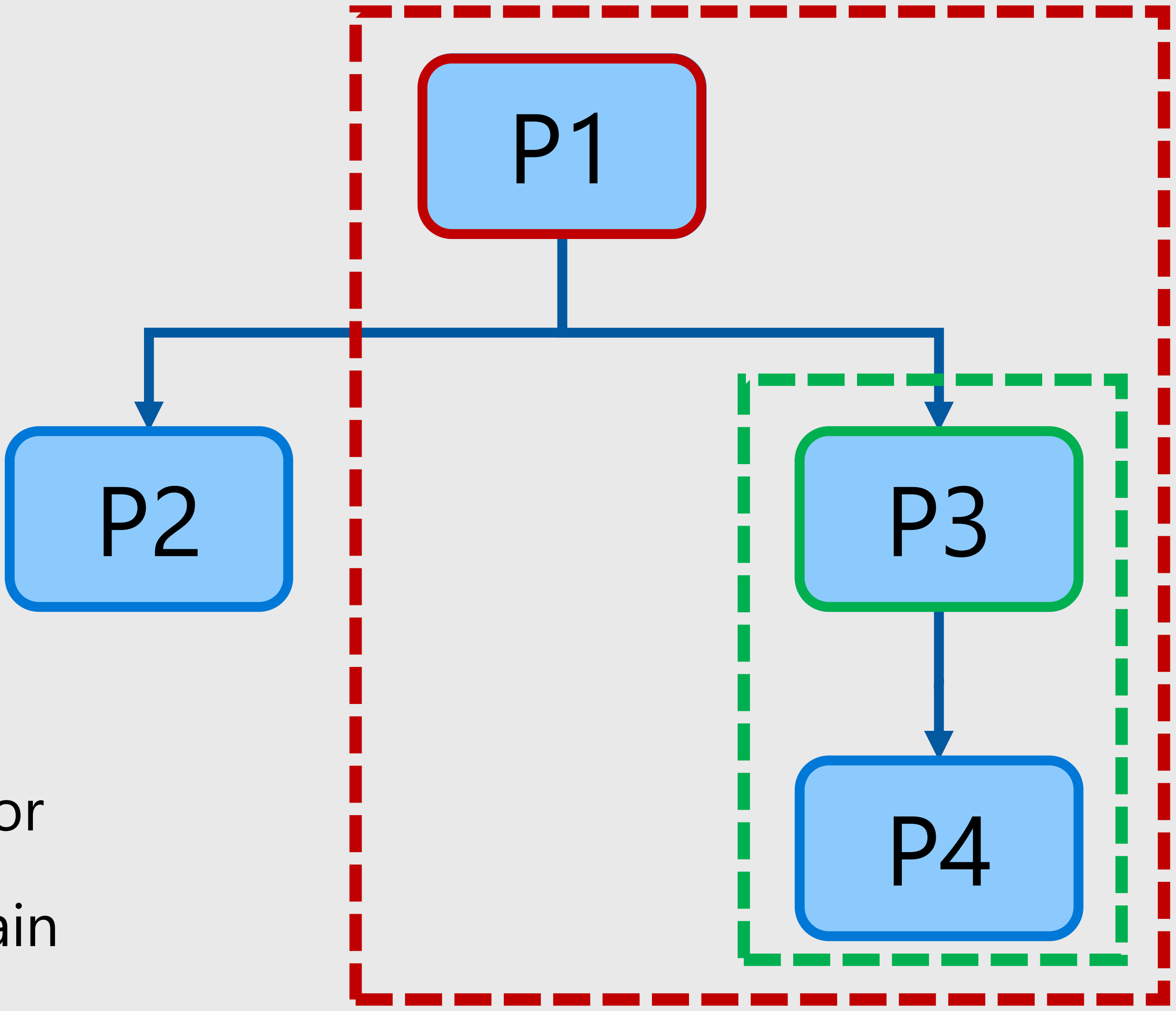
Inherit parent policies

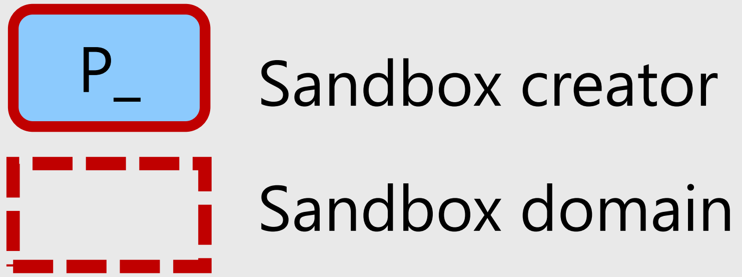
A sandbox can only drop more accesses.

Forbid access to parent or sibling sandboxes

Introspection (i.e., ptrace) of processes not in in a child sandbox (or the same sandbox) is forbidden.

Automatic hierarchy restrictions





Demo

Simple sandbox manager using Landlock

https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/samples/landlock/sandboxer.c

Interesting development properties

Thanks to security policy compositions:

Lockless concurrent development: no bottleneck because of one global policy.

It is easier to maintain a set of **small policies**.

Because policies can be embedded within the code:

Tailored to an application: developers know how it works and what is required.

Can be kept in sync with evolving business logics **over time**.

Can dynamically be built according to an application **configuration** and then adapt to configuration changes.

Can be **tested** like another (unprivileged) user space feature.

Landlock vocabulary

Object

Kernel resource: file, file hierarchy, process…

Action

Action on an object: list the content of a directory, write to a file, create a pipe…

Rule

Set of actions on an object

Ruleset

Set of rules

How to use Landlock?

Three system calls:

- landlock_create_ruleset()
- landlock_add_rule()
- landlock_restrict_self()

Step 1: Create a ruleset

```
int ruleset_fd;
struct landlock_ruleset_attr ruleset_attr = {
    .handled_access_fs =
        LANDLOCK_ACCESS_FS_EXECUTE |
        LANDLOCK_ACCESS_FS_WRITE_FILE |
        […]
        LANDLOCK_ACCESS_FS_MAKE_REG,
};

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

Step 2: Add rules

```
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");
```

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);
```

Developer tools

glibc (2.34) and soon in musl libc

strace (5.13) support for debugging

Go library: https://github.com/landlock-lsm/go-landlock

Rust library: https://github.com/landlock-lsm/rust-landlock

Kernel compatibility

Problem

Application developers may not be aware of the kernel on which their application will run.

It is better for users to implement a **best-effort security**: use available sandboxing features as much as possible.

Landlock will gain new features over time.

Solution

Design Landlock syscalls as backward and forward compatible with previous and future kernel versions.

Backward compatibility

Problem

Running applications developed for a new kernel, on an old kernel.

Solution

Ability to probe the kernel to get the Landlock ABI version, which indicates the sandboxing features supported by the running kernel.

Future-proofness

Problem

Running applications developed for an old kernel, on a new kernel.

Solution

The Linux kernel ABI must always be compatible with previous versions.

Use extensible structs as syscall argument to enable **flexible addition of future features**: a struct can be larger than expected but the unknown fields must contain zeros.

Use optional flags.

```
landlock_create_ruleset(&ruleset_attr, sizeof(ruleset_attr), 0);
```

Roadmap (kernel-side)

Short term

- Improve kernel performance for the current features.
- Add the ability to change the parent directory of files (see current Landlock limitations).

Medium term

- Add audit features to ease debugging.
- Extend filesystem access-control types to address the current limitations.
- Add the ability to follow a deny listing approach, which is required for some use cases.

Long term

- Add minimal network access-control types.
- Add the ability to create (file descriptor) capabilities compatible with Capsicum.

Wrap-up

Landlock empowers developers to harden their applications and protect users' data: manage tailored and composable policies.

New kernel releases will lift most current limitations, improve performance, and bring new sandboxing features.

High-level Go and Rust libraries.

Questions: landlock@lists.linux.dev

Resources: https://landlock.io