Microsoft

Linux sandboxing with Landlock

Overview & workshop

Mickaël Salaün – kernel maintainer

2025-01-29

How does a data breach happen?

bugs

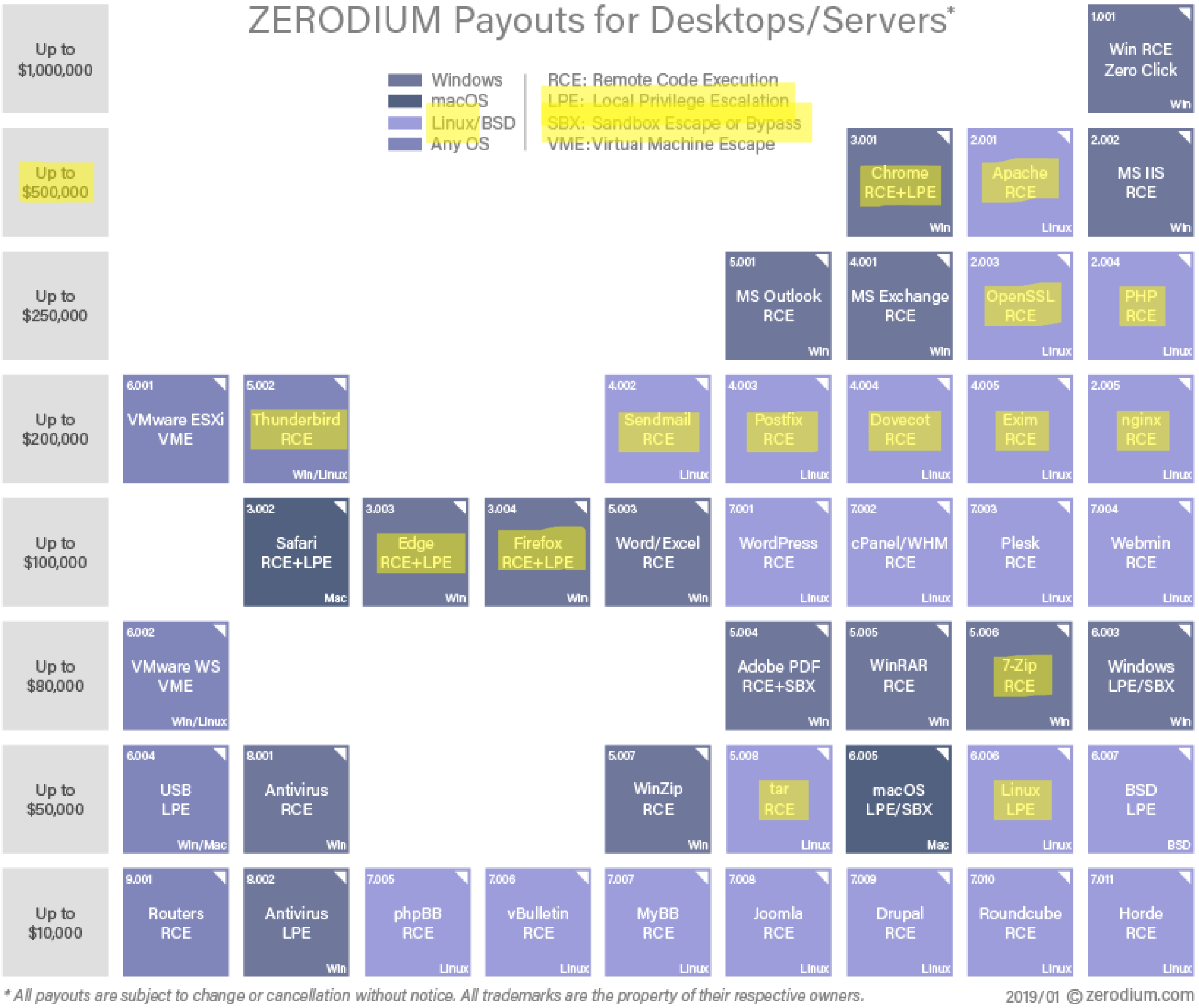
+

malicious actors

=

exploited vulnerabilities

Real-life exploits



ZERODIUM Payouts for Desktops/Servers*

- ▨ Windows
- ▨ macOS
- ▨ Linux/BSD
- ▨ Any OS

RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass
VME: Virtual Machine Escape

Up to $1,000,000									1.001 Win RCE Zero Click Win
Up to $500,000							3.001 Chrome RCE+LPE Win	2.001 Apache RCE Linux	2.002 MS IIS RCE Win
Up to $250,000						5.001 MS Outlook RCE Win	4.001 MS Exchange RCE Win	2.003 OpenSSL RCE Linux	2.004 PHP RCE Linux
Up to $200,000	6.001 VMware ESXi VME	5.002 Thunderbird RCE Win/Linux			4.002 Sendmail RCE Linux	4.003 Postfix RCE Linux	4.004 Dovecot RCE Linux	4.005 Exim RCE Linux	2.005 nginx RCE Linux
Up to $100,000		3.002 Safari RCE+LPE Mac	3.003 Edge RCE+LPE Win	3.004 Firefox RCE+LPE Win	5.003 Word/Excel RCE Win	7.001 WordPress RCE Linux	7.002 cPanel/WHM RCE Linux	7.003 Plesk RCE Linux	7.004 Webmin RCE Linux
Up to $80,000	6.002 VMware WS VME Win/Linux					5.004 Adobe PDF RCE+SBX Win	5.005 WinRAR RCE Win	5.006 7-Zip RCE Win	6.003 Windows LPE/SBX Win
Up to $50,000	6.004 USB LPE Win/Mac	8.001 Antivirus RCE Win			5.007 WinZip RCE Win	5.008 tar RCE Linux	6.005 macOS LPE/SBX Mac	6.006 Linux LPE Linux	6.007 BSD LPE BSD
Up to $10,000	9.001 Routers RCE	8.002 Antivirus LPE Win	7.005 phpBB RCE Linux	7.006 vBulletin RCE Linux	7.007 MyBB RCE Linux	7.008 Joomla RCE Linux	7.009 Drupal RCE Linux	7.010 Roundcube RCE Linux	7.011 Horde RCE Linux

* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

2019/01 © zerodium.com

Pragmatic statements

1. An innocuous and trusted process can **become malicious during its lifetime** because of bugs exploited by attackers.
2. There are multiple and **different levels of trust (TCB)** and different **consequences** in case of a breach: system, user, app data…

Agenda

1. Secure development
2. Sandboxing
3. Sandboxing on Linux
4. Landlock status
5. Landlock properties
6. Landlock access control
7. Sandboxing with Landlock
8. Workshop setup
9. Let’s patch ImageMagick!
10. Compatibility

Securing developments

How to protect an application?

Reactive solutions

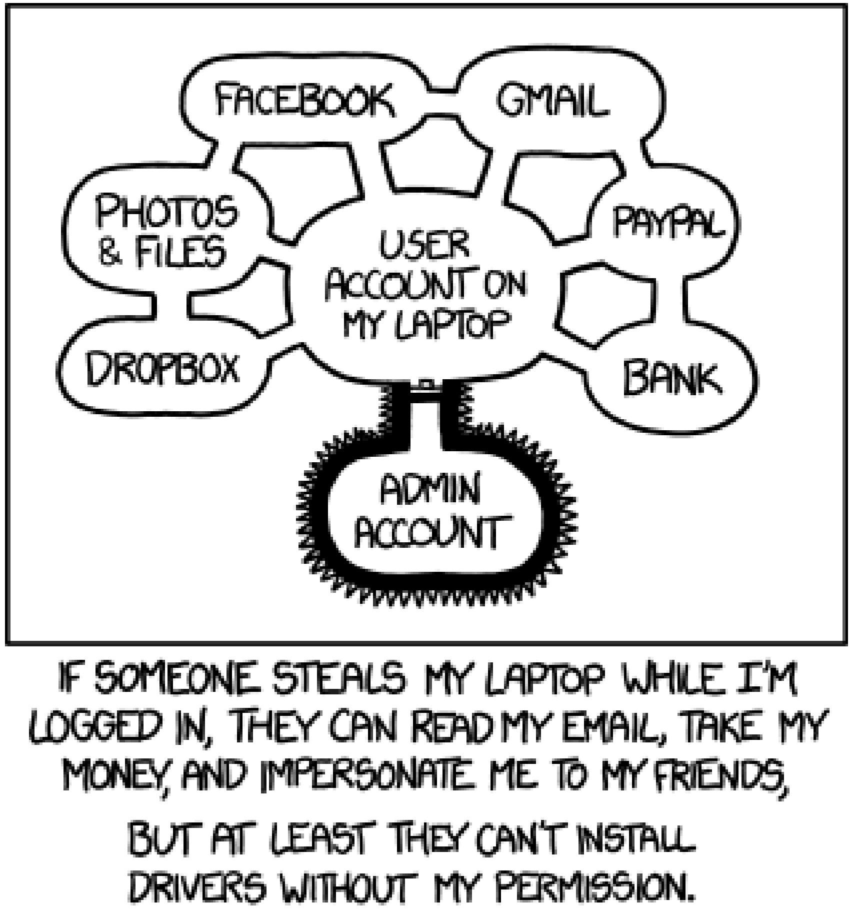
Fix bugs quickly and push updates widely

How to protect an application?

Proactive solutions

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

Protect 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

Sandboxing

What is sandboxing?

“A **restricted**, controlled **execution environment** that prevents potentially malicious software […] from accessing any system resources except those for which the software is authorized.”

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
- Interactions

Safe security mechanism

Principle of least privilege

- No privileged accounts or services
- No SUID binaries

Innocuous access control

- Only increase restrictions

Protecting against bypasses

- Each process should be protected from less-privileged ones

Non-Linux systems

Main sandbox mechanisms:

- XNU Sandbox (iOS)
- Pledge and Unveil (OpenBSD)
- Capsicum (FreeBSD)
- AppContainer (Windows)

Sandboxing on Linux

Virtual machine

Pros

- Duplicate the whole system and then mitigates its exposure

Cons

- Shipping a VM instead of an installer is a hard sell because of size, overhead and complexity
- Does not provide an access control system

Container

Pros

- Well known developer tool
- Lighter than a VM

Cons

- May increase the attack surface and comes with its own vulnerabilities: namespaces and embed dependencies
- May provide some coarse-grained control for file access, but not native to apps/services: increased configuration

AppArmor, SELinux, Smack, or Tomoyo

Pros

- Real access control systems

Cons

- Security policy is system-wide and cannot be embedded in apps/services: complex and static configuration

BPF LSM

Pros

- Dynamic security policies

Cons

- Security policy is system-wide and cannot be embedded in apps/services: complex eBPF programs
- Difficult to deal with files

seccomp-bpf

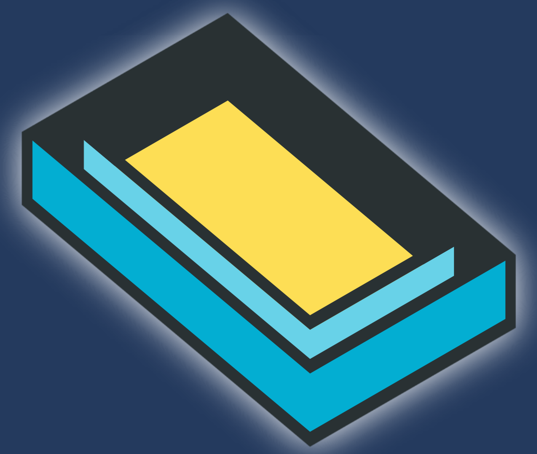
Pros

- Dynamic security policies
- Reduces the kernel attack surface
- Embeddable in apps/services: unprivileged

Cons

- Not an access control system: cannot identify files nor other kernel semantic
- Fixed set of syscalls: update issues
- Scoped to a set of processes

Landlock



Pros

- Real access control system
- Dynamic security policies
- Embeddable in apps/services: unprivileged

Cons

- Scoped to a set of processes

Candidates for a sandboxing mechanism

	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

Landlock status

The Linux kernel development

One of the largest and most active free software projects in existence.

New release every ~9 weeks

26+ million single lines of code

Development statistics for Linux 6.12:

- 2000+ developers, including 300+ new contributors
- 13000+ commits

Landlock development

Maintainer: Mickaël Salaün

Reviewer: Günther Noack

Main contributors: Konstantin Meskhidze, Jeff Xu, Ivanov Mikhail, Jann Horn, Tahera Fahimi

History

1. Initial RFC (Mar. 2016)
2. 34 patch series with different designs: seccomp, eBPF, cgroups…
3. Merged in Linux 5.13 (Apr. 2021)

Landlock in numbers

Single lines of code (Linux 6.12):

- Kernel: ~2500
- Tests: ~7160 (without LTP)

Tests coverage: 92%

Fuzzing coverage with syzkaller: 72%

Documentation: 29 pages

Article: 34 pages

Linux distributions

Most distros support Landlock by default:

- Arch Linux
- Ubuntu
- Debian
- Fedora
- chromeOS
- WSL2
- Azure Linux
- Gentoo
- Flatcar
- RHEL (WIP)…

Container runtimes

Most container runtimes supporting Landlock:

- Docker
- Podman
- runc
- LXC
- systemd-nspawn

Landlock helpers

Examples of sandbox tools:

- setpriv
- Minijail
- Firejail

Examples of sandbox libraries:

- Landlock Rust crate
- Landlock Go library
- Minijail
- Pledge for Linux

Landlocked apps

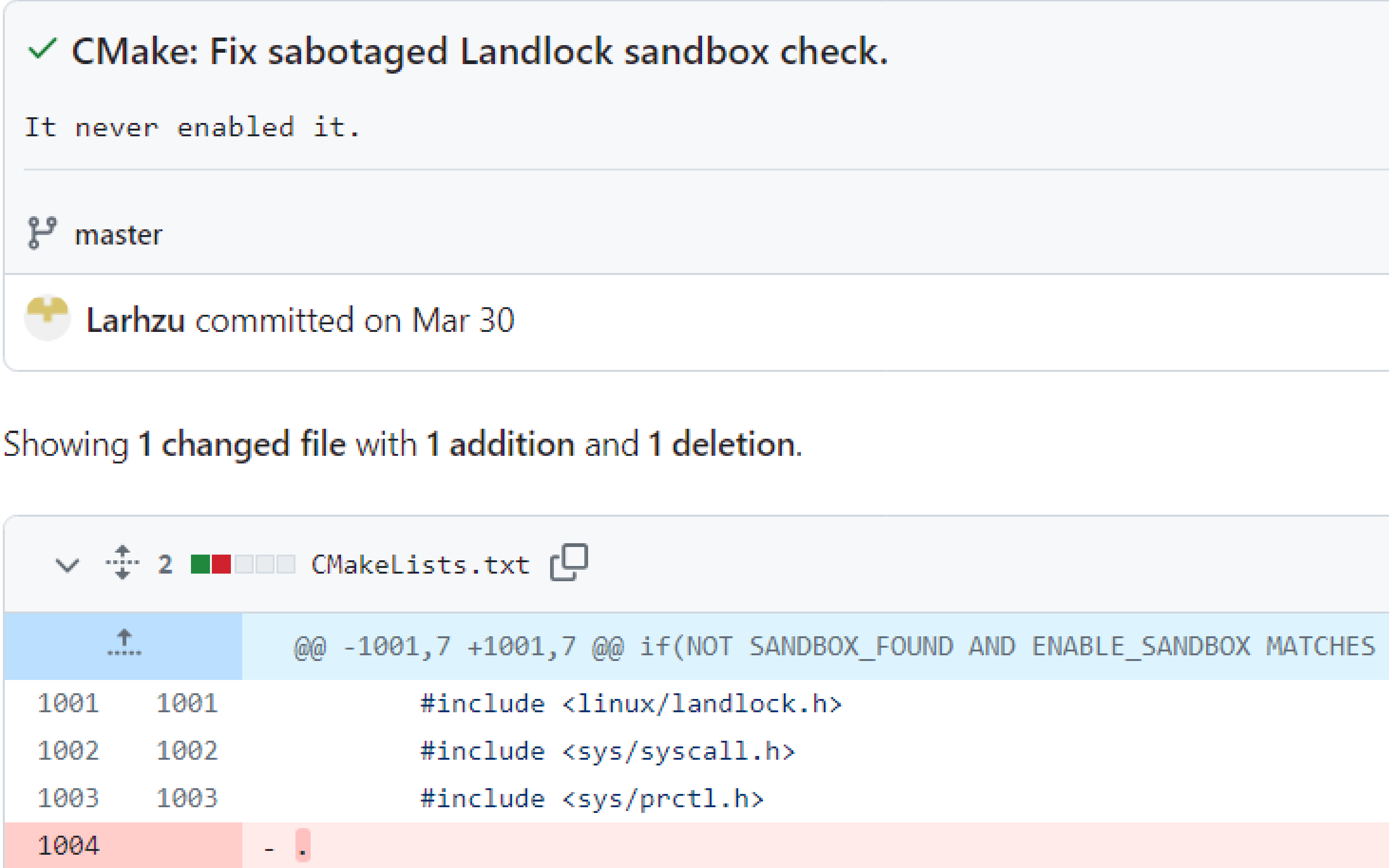
Examples of sandboxed apps:

- Zathura (document viewer)
- Pacman (package manager)
- Cloud Hypervisor (VM monitor)
- Suricata (network IDS)
- Polkadot (blockchain SDK)
- wireproxy (Wireguard client)
- GNOME LocalSearch (search engine)
- XZ Utils (archive manager)

Getting noticed by attackers too!

Landlock support in XZ Utils:

- 5.6.0 (2024-02-24) ✓
- 5.6.1 (2024-03-09) ✗
- 5.6.2 (2024-05-29) ✓



✓ CMake: Fix sabotaged Landlock sandbox check.

```
It never enabled it.
```

master

Larhzu committed on Mar 30

Showing **1 changed file** with **1 addition** and **1 deletion**.

```
@@ -1001,7 +1001,7 @@ if(NOT SANDBOX_FOUND AND ENABLE_SANDBOX MATCHES
1001  1001          #include <linux/landlock.h>
1002  1002          #include <sys/syscall.h>
1003  1003          #include <sys/prctl.h>
1004        - .
```

Try Landlock

```
# WARNING: The "sandboxer" is a demonstration program,
# not a tool with a stable interface.

$ cargo install landlock --examples

$ sandboxer
```

Landlock properties

Use case #1

Untrusted applications: protect from potentially malicious third-party code.

Candidates:

- Container runtimes
- Init systems

Use case #2

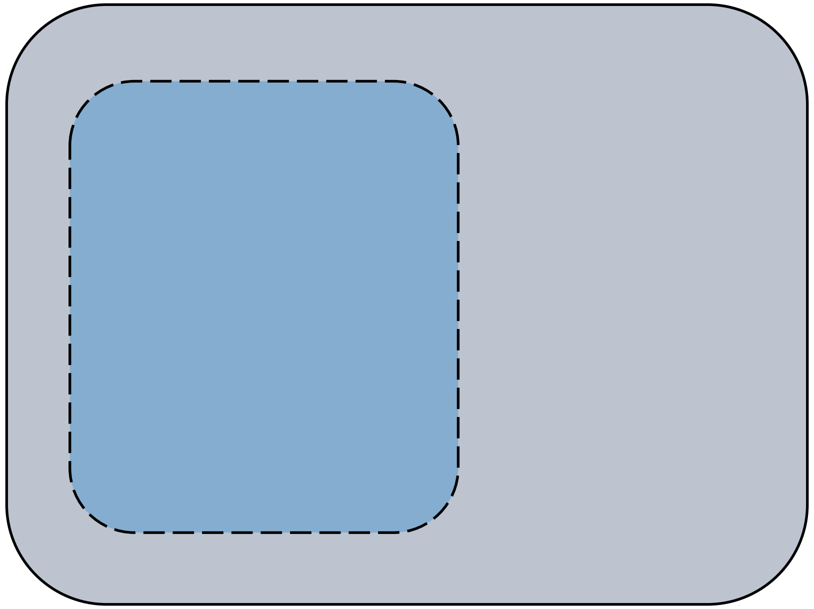
Exploitable bugs in trusted applications: protect from vulnerable code maintained by developers.

Candidates:

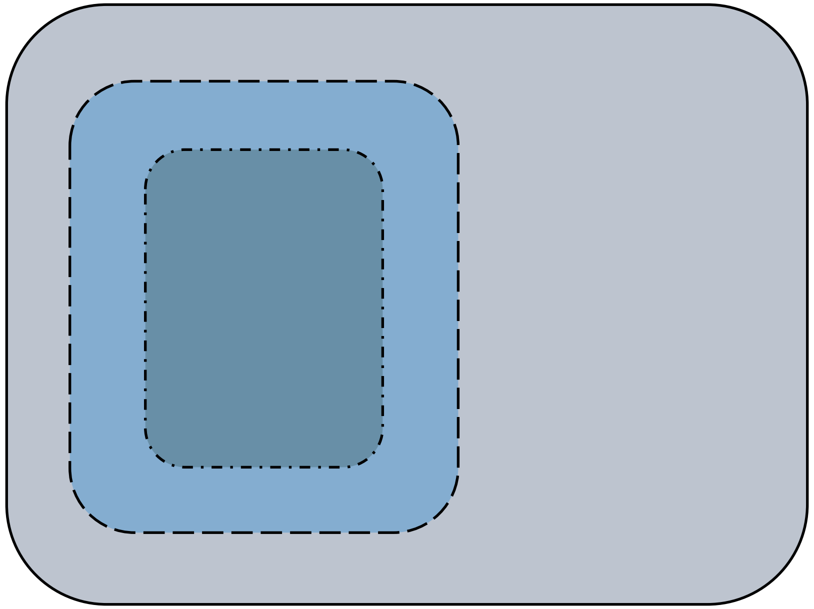
- Parsers: archive tools, file format conversion, renderers…
- Web browsers
- Network and system services

Dynamic policy composition

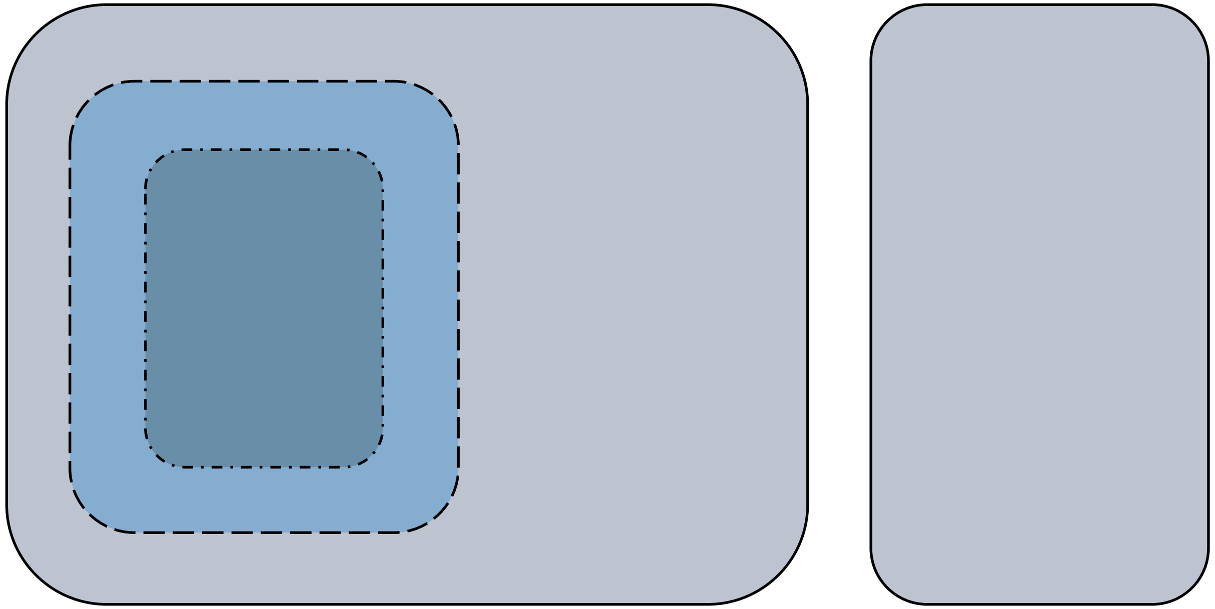
Dynamic policy composition



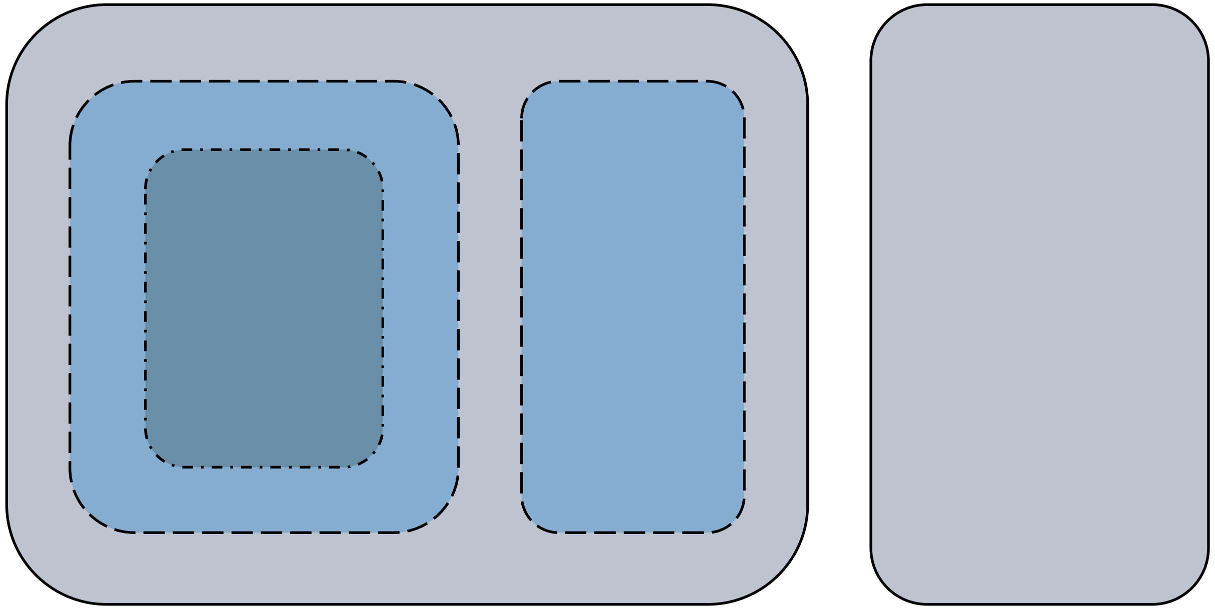
Dynamic policy composition



Dynamic policy composition



Dynamic policy composition

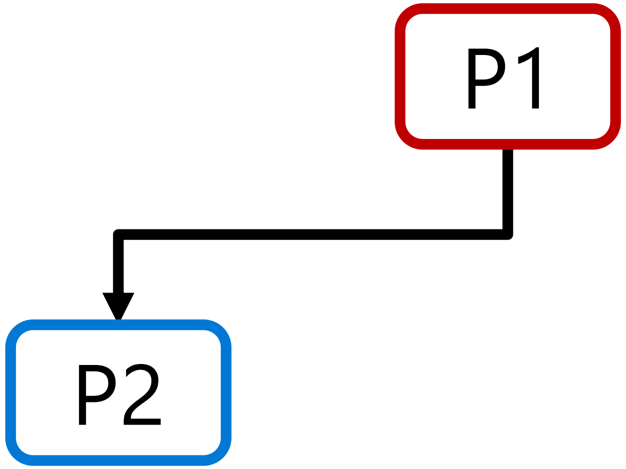


Sandbox policies hierarchy

P_ Sandboxed process

⬚ Sandbox domain

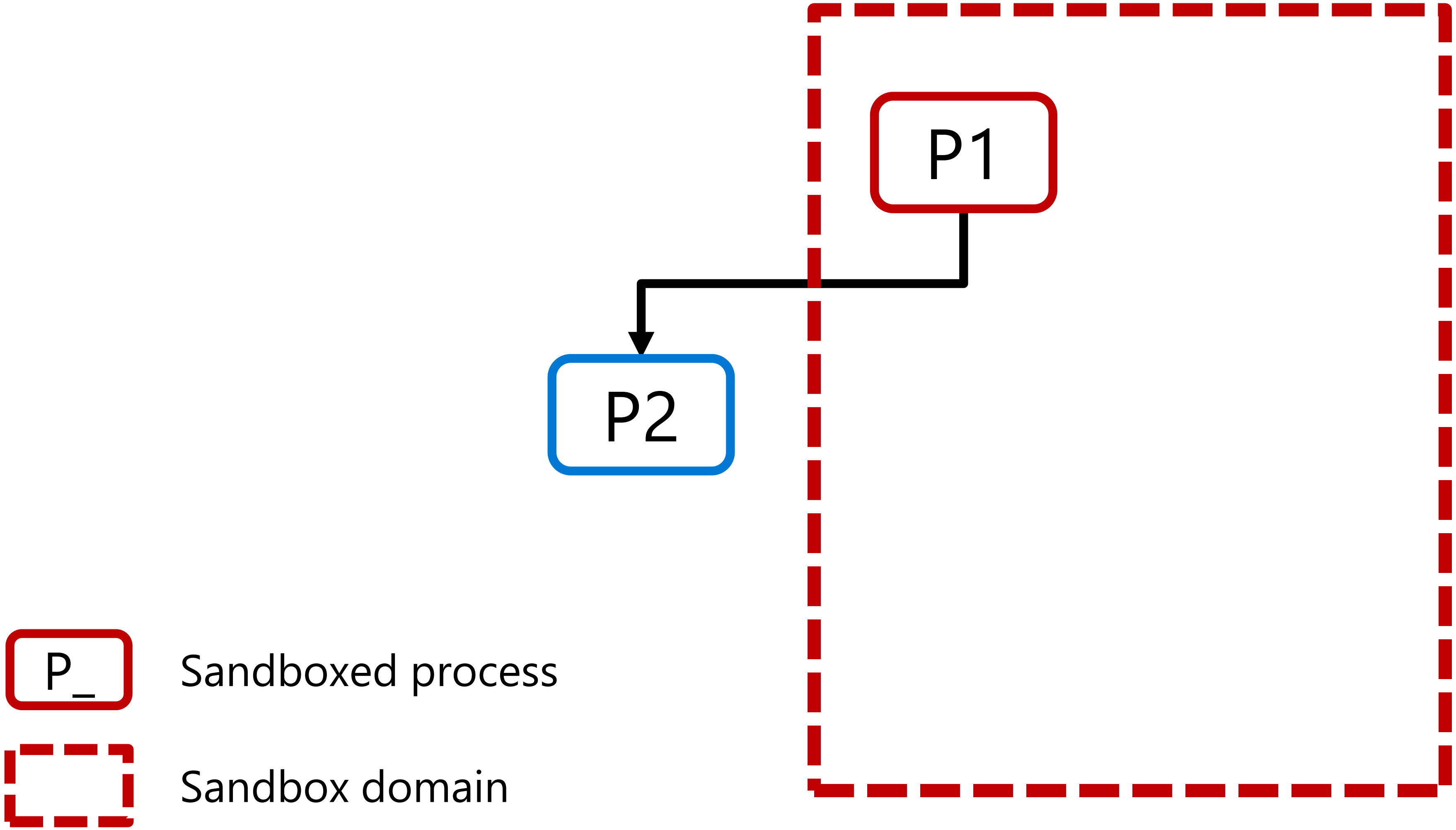
Sandbox policies hierarchy



P_ Sandboxed process

⬚ Sandbox domain

Sandbox policies hierarchy



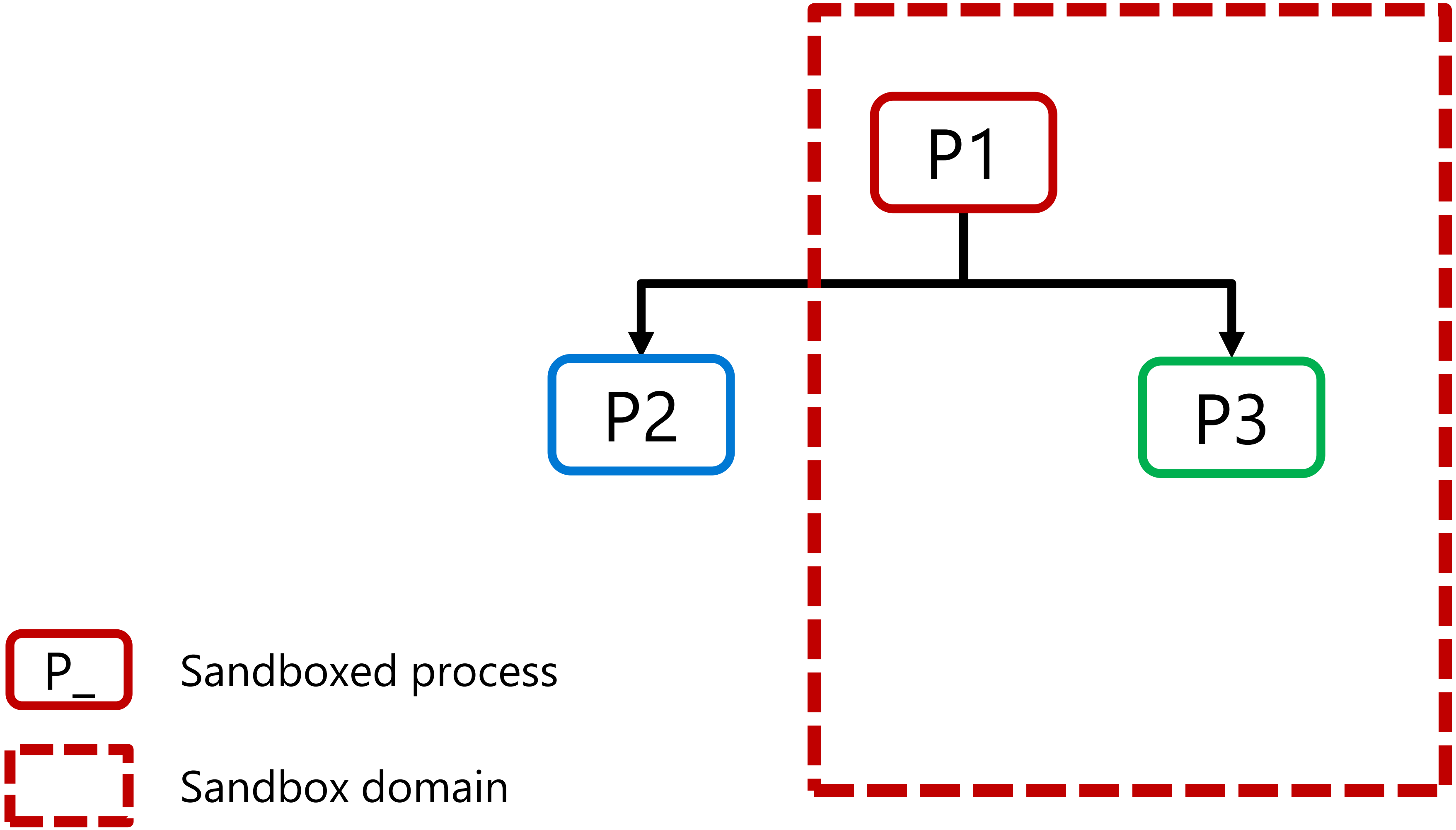
P1

P2

P_ Sandboxed process

Sandbox domain

Sandbox policies hierarchy



P1

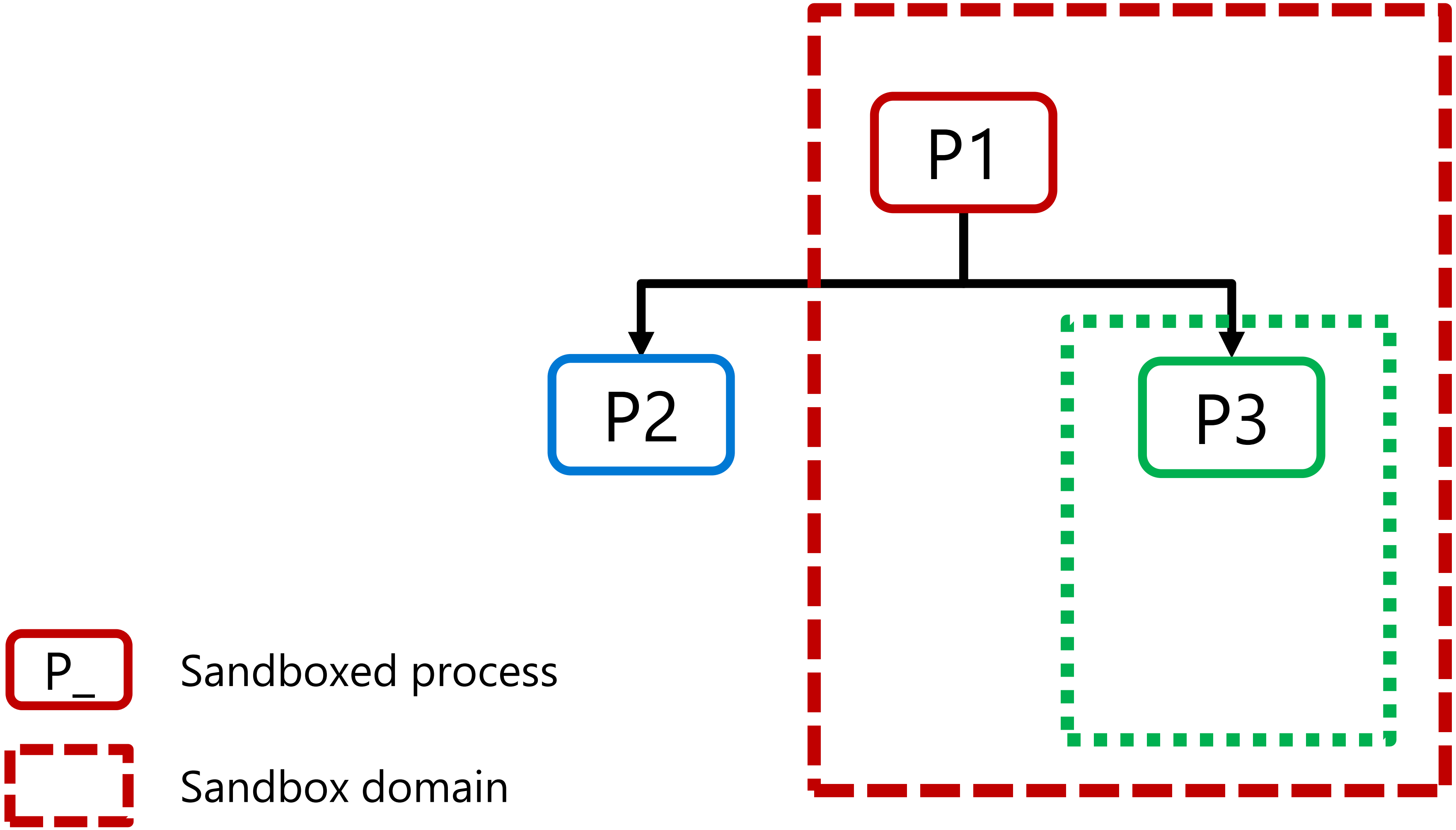
P2

P3

P_ Sandboxed process

Sandbox domain

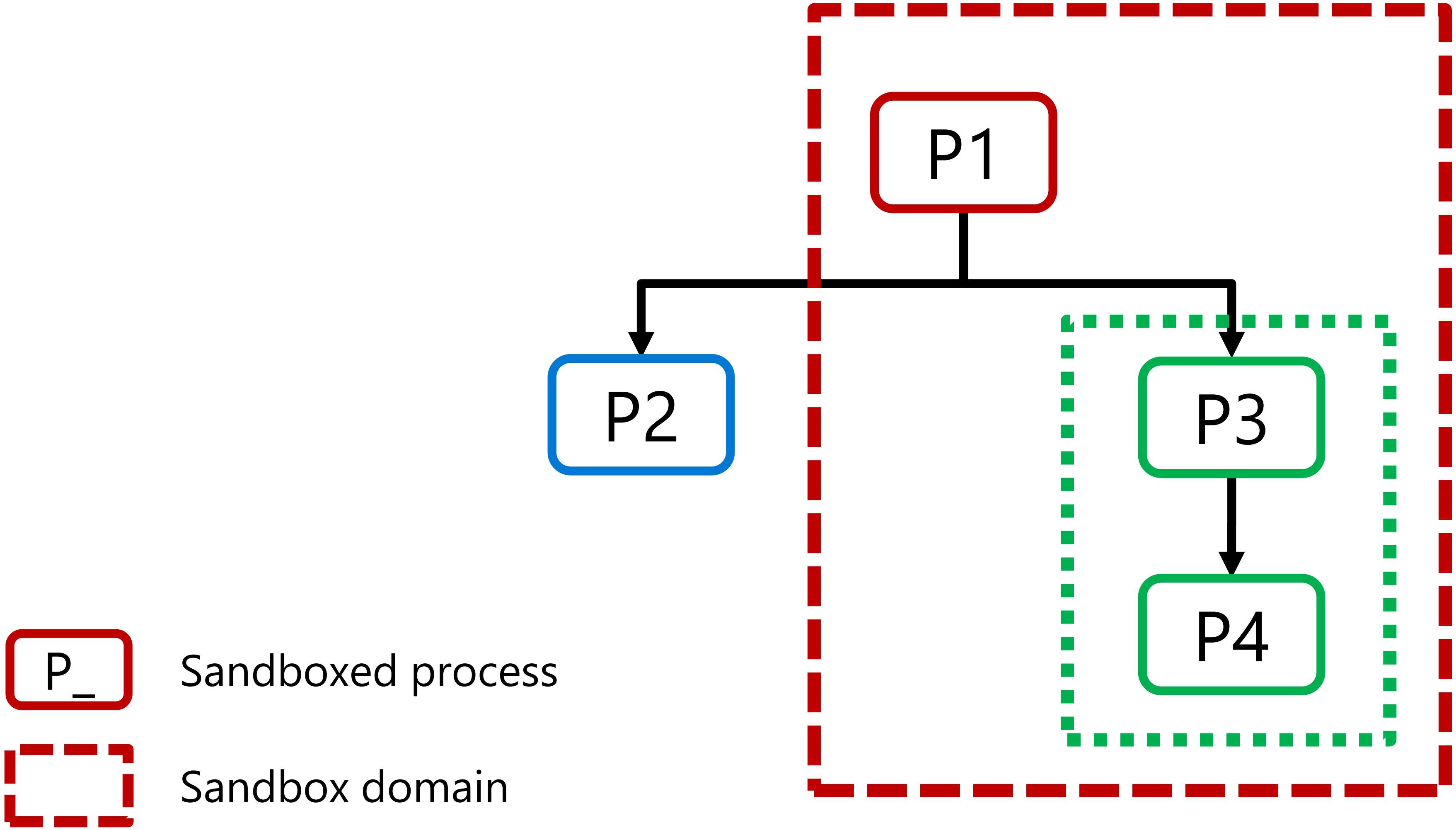
Sandbox policies hierarchy



Sandboxed process

Sandbox domain

Sandbox policies hierarchy



P1

P2

P3

P4

P_ Sandboxed process

Sandbox domain

Useful development properties

Embedded policies: **testable with a CI** and always synchronized with **app semantic**

Set of small policies: **easier to maintain** and audit

Composable policies: **lockless concurrent policy development**

Well-defined backward compatibility with ABI versions: stable and **consistent results**

How does Landlock work?

Restrict ambient rights according to the **kernel semantic** (e.g., global filesystem access) for a set of processes, thanks to **3 dedicated syscalls**.

Security policies are inherited by all new children processes.

A one-way set of restrictions: cannot be disabled once enabled.

Landlock access control

Current access control

Implicit restrictions

- Process impersonation (e.g., ptrace)
- Filesystem topology changes (e.g., mounts)

Explicit access rights

- Filesystem
- Networking
- Signaling
- Abstract unix socket

IPC scoping

Scope sandboxes:

- Connect to abstract UNIX sockets
- Send signals

Current networking access rights

- Connect to a TCP port
- Bind to a TCP port

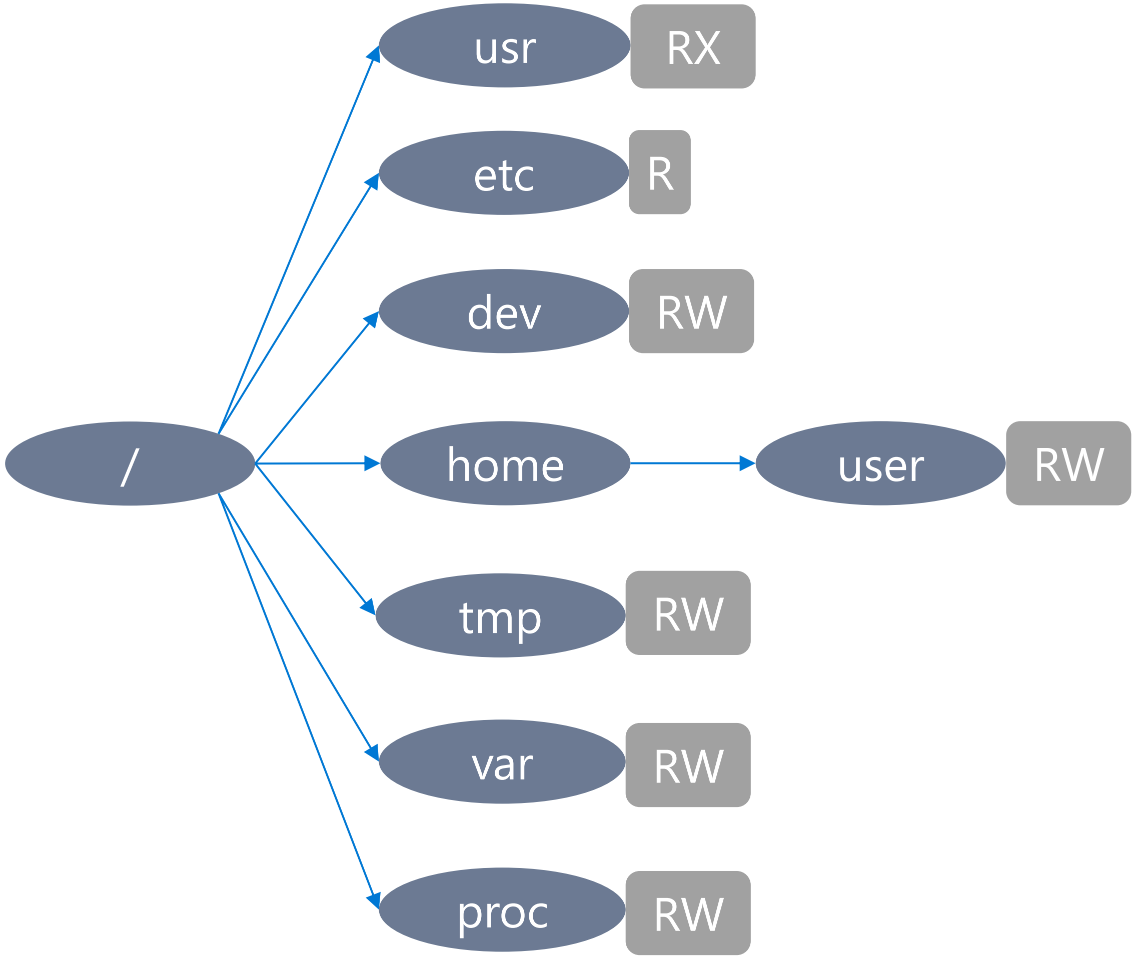
Current filesystem access rights

- Execute, read or write to a file
- List a directory or remove files
- Create files according to their type
- Rename or link files
- Send IOCTL commands to devices

Example of filesystem policy composition

Example of filesystem policy composition

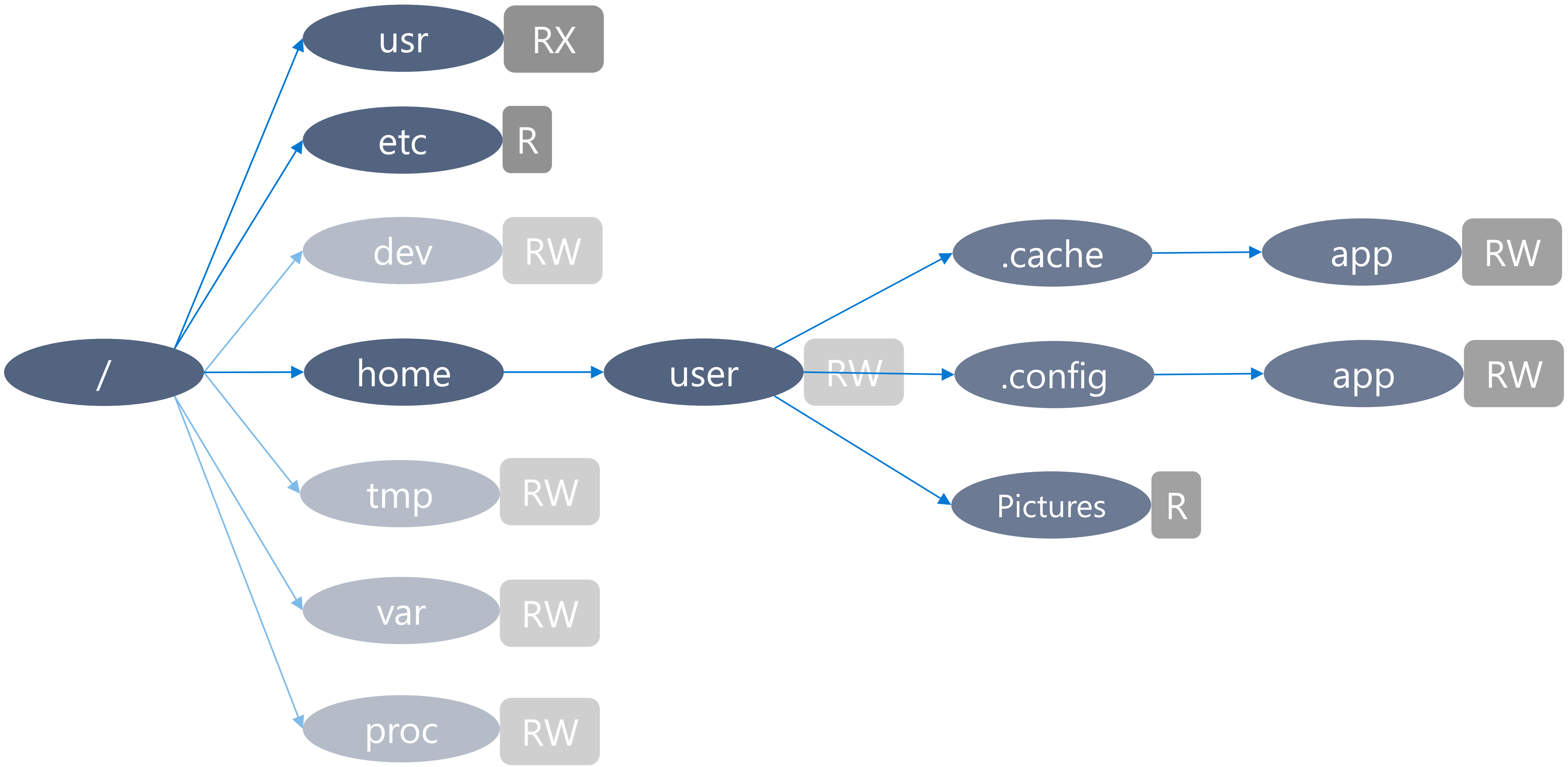
1st layer



- R Read
- W Write
- X eXecute

Example of filesystem policy composition

2nd layer



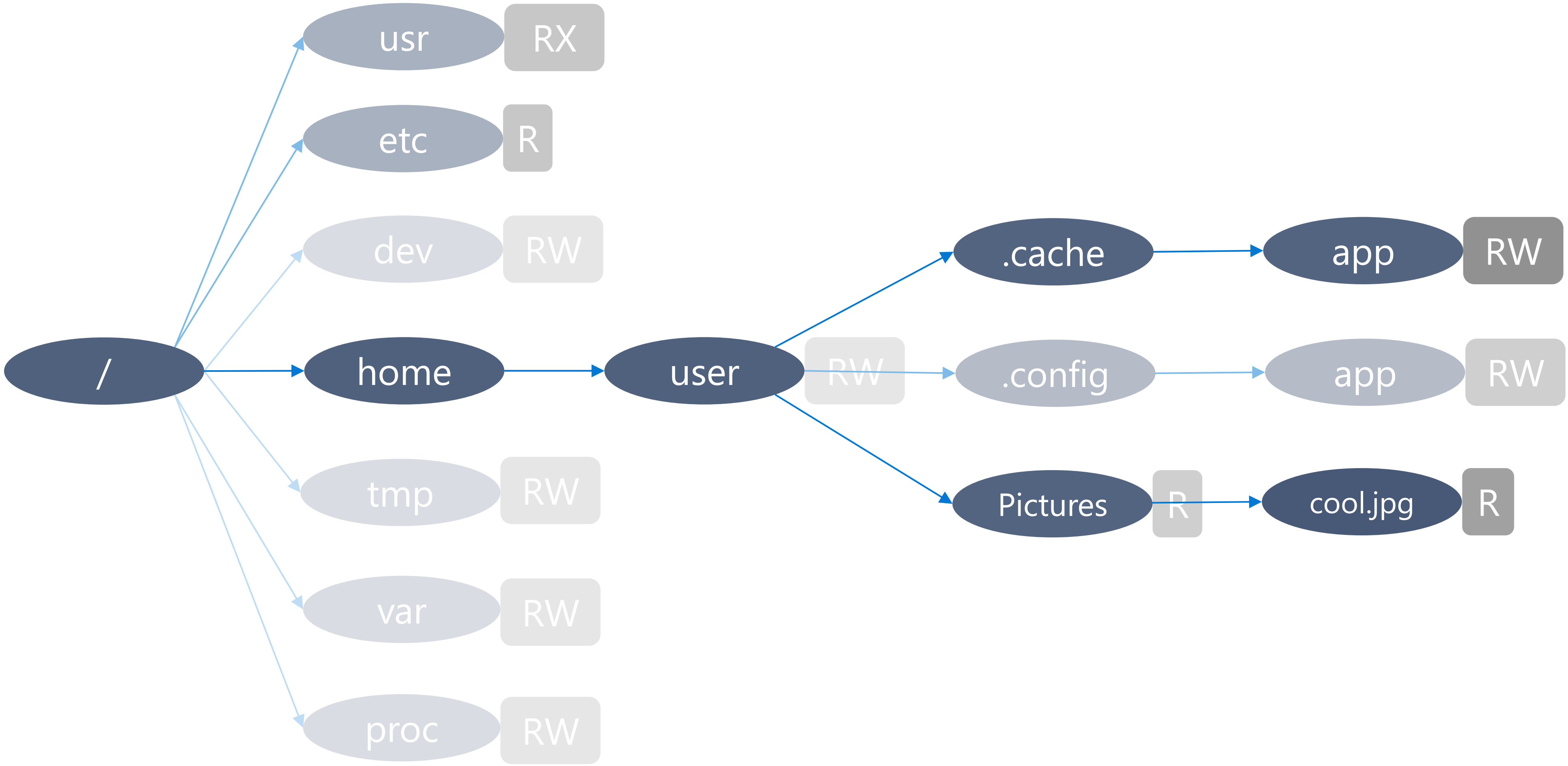
R Read

W Write

X eXecute

Example of filesystem policy composition

3rd layer



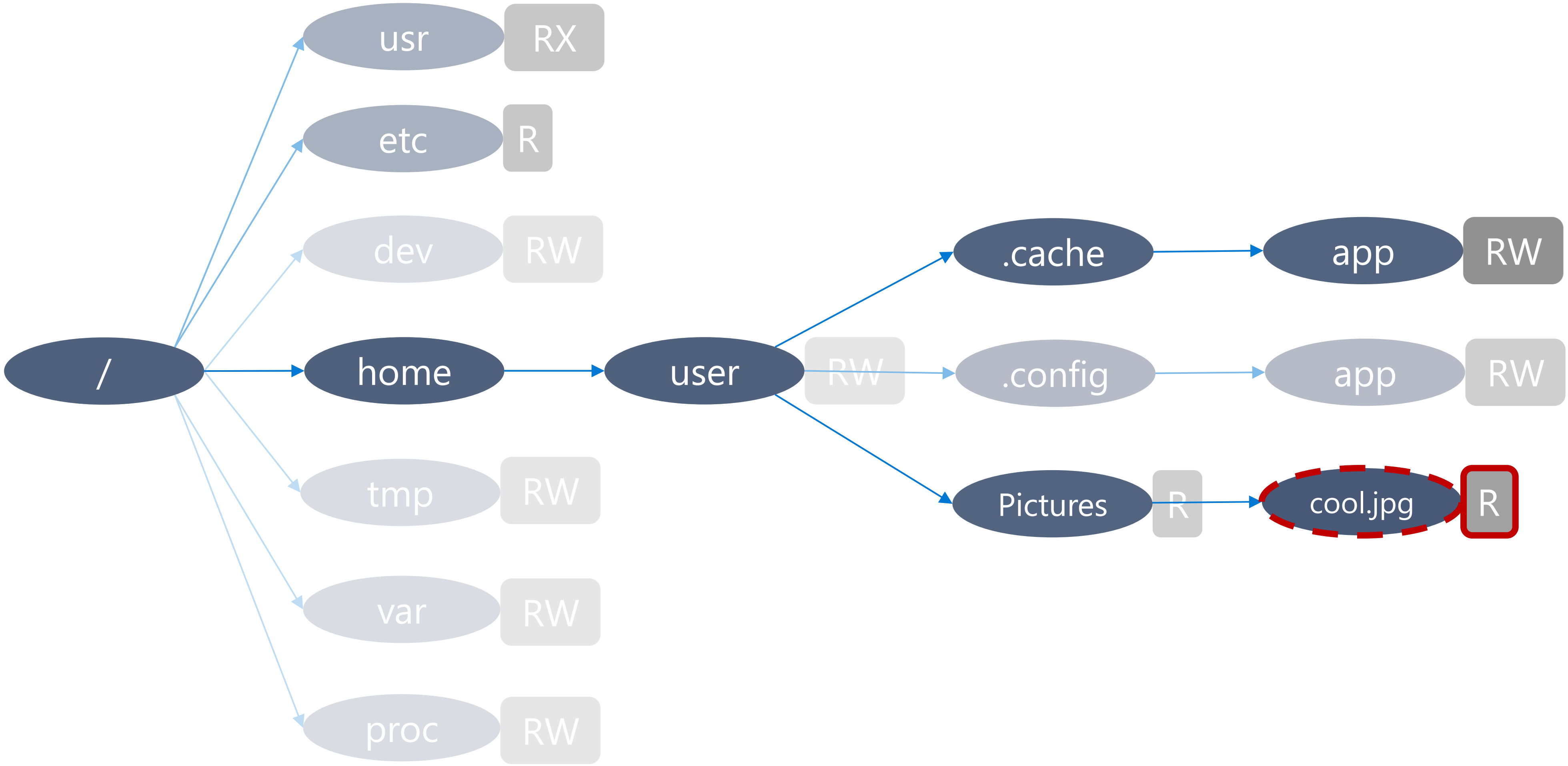
R Read

W Write

X eXecute

Example of filesystem policy composition

3rd layer ✓



R Read

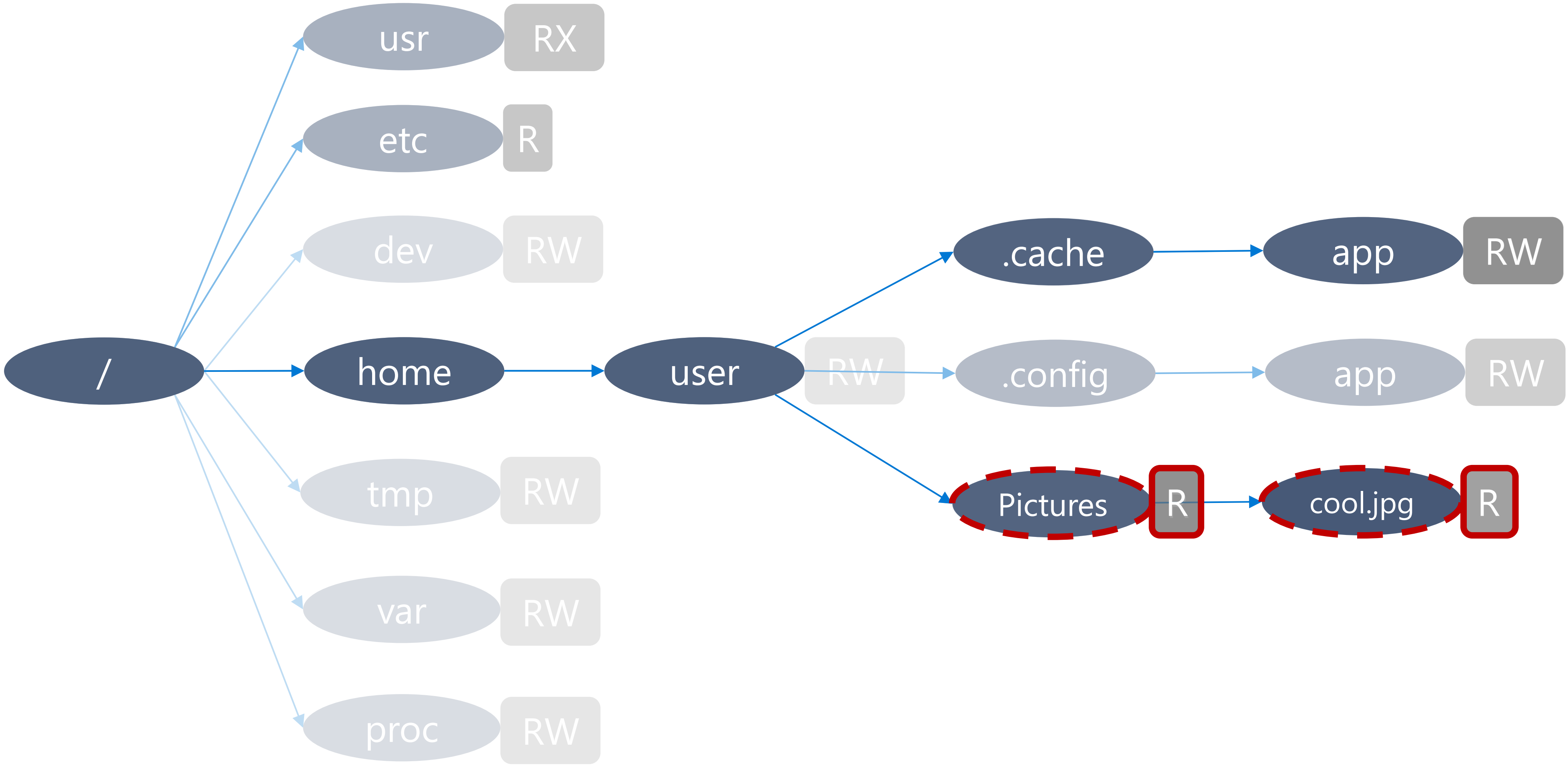
W Write

X eXecute

Example of filesystem policy composition

3rd layer ✓

2nd layer ✓



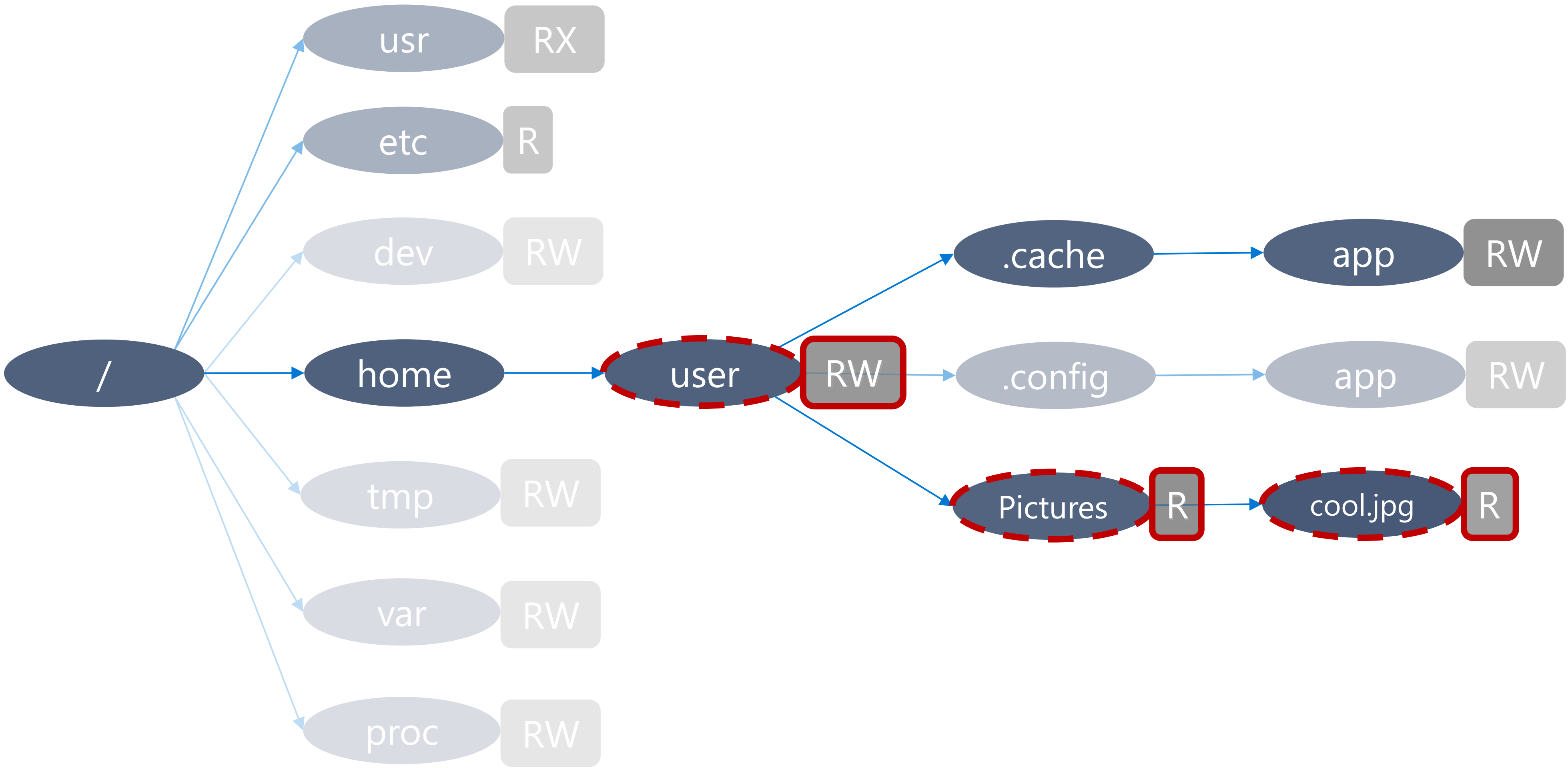
R	Read
W	Write
X	eXecute

Example of filesystem policy composition

3rd layer ✓

2nd layer ✓

1st layer ✓



R Read

W Write

X eXecute

Sandboxing with Landlock

How to patch an application?

1. Define the threat model: which data is trusted or untrusted?
2. Identify the complex parts of the code: where there is a good chance to find bugs?
3. Identify and patch the configuration handling to infer a security policy.
4. Identify and patch the most generic places to enforce the security policy for the rest of the lifetime of the thread.

Application compatibility in a nutshell

Forward compatibility: kernel

Backward compatibility: responsibility of application developers

Each new Landlock feature increments the ABI version, which is useful to leverage available features in a **best-effort security** approach.

Will see more at the end of this talk…

Landlock ABI versions

1. Linux 5.13: Initial set of FS access rights
2. Linux 5.19: Rename and link
3. Linux 6.2: Truncation
4. Linux 6.7: TCP connect and bind
5. Linux 6.10: IOCTL for devices
6. Linux 6.12: Signal and abstract UNIX socket

Landlock interface (in C and Rust)

Step 1: Check backward compatibility

```
int abi = landlock_create_ruleset(NULL, 0, LANDLOCK_CREATE_RULESET_VERSION);
if (abi < 0)
    return 0;
```

Step 2: Create a ruleset

```
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");
```

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

Step 3: 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");
```

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

Step 4: 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::default()
    .handle_access(make_bitflags!(
        AccessFs::{Execute | WriteFile}))?
    .create()?
    .add_rule(
        PathBeneath::new(PathFd::new("/")?)
        .allow_access(AccessFs::Execute)
    )?
    .restrict_self()?
```

Full example in C

Full example in Rust

Workshop setup

VM setup

See https://github.com/landlock-lsm/workshop-imagemagick

If you already cloned the repository:

```
git pull
vagrant up
vagrant ssh
```

Connect to the VM

```
# Once set up, take a snapshot and log in

vagrant snapshot push
vagrant ssh


# We can now also use virt-manager to connect to the VM
```

Steps done by the VM provisioning

1. Set up the build environment
2. Build a vulnerable version of ImageMagick

Let’s patch ImageMagick!

ImageMagick

Pretty common set of tools to transform or display pictures: parse a lot of file formats

Use cases: CLI tool or (web) server

Attack scenario

CVE-2016-3714/ImageTragick: insufficient shell characters filtering that can lead to (potentially remote) code execution.

Let’s say we have a vulnerable version, not necessarily this one. For this workshop we use an old and vulnerable (long-been-fixed) ImageMagick version.

Sandboxing this kind of tool can help mitigate the impact of such vulnerability: e.g., deny access to secret files

Agenda

1. Test an exploit
2. Find the sweet spot to restrict the process
3. Patch + build + test

Test exploit with vulnerable version

```
# Convert from one image format to another

convert /vagrant/exploit/malicious.mvg /tmp/out.png

# Solution patches are available in /vagrant/imagemagick-patches/*.patch
```

Main steps to patch

1. Declare the Landlock syscalls
2. Find what we want to sandbox and where it would make sense
3. Create a ruleset
4. Add static rules
5. Add dynamic rules
6. Restrict the task before potentially-harmful computation

Patch ImageMagick 1/9

```
# 1/ Go to the source directory

cd ~/src/ImageMagick-6.9.3-8
```

Patch ImageMagick 2/9

```
# 3/ Import Landlock syscall stubs and access right groups

cp /vagrant/sandboxer.c magick/landlock.h
vim magick/landlock.h

git add -A
git commit

# 4/ Look at the system’s Landlock definitions and types (updated with up-to-date 6.12 headers)

vim /usr/include/linux/landlock.h
```

Patch ImageMagick 3/9

```
   # 5/ Look at the convert code and find a sweat spot for sandboxing

   vim wand/convert.c

   # Imagemagick doesn’t have a clear separation between argument parsing and their evaluation: we need to patch the loop parsing arguments.

   # 6/ Include landlock.h and create the ruleset in ConvertImageCommand()

   (void) CopyMagickString(image_info->filename,filename,MaxTextExtent);

+  const struct landlock_ruleset_attr ruleset_attr = {
+      .handled_access_fs = ACCESS_FS_ROUGHLY_READ | ACCESS_FS_ROUGHLY_WRITE,
+  };
```

0003-WORKSHOP-Create-a-ruleset.patch

Build and test the patched ImageMagick

```
# Regularly build and check convert

make

# Test conversion (the convert tool points to ./utilities/convert)

convert /vagrant/exploit/malicious.mvg /tmp/out.png

# Debug

strace convert /vagrant/exploit/malicious.mvg /tmp/out.png
```

Patch ImageMagick 4/9

```
# 7/ Create the ruleset

int ruleset_fd = landlock_create_ruleset(&ruleset_attr, sizeof(ruleset_attr), 0);

# 8/ Check for errors and log them

if (ruleset_fd < 0) {
    perror("LANDLOCK: Failed to create a ruleset");
    return MagickFalse;
}

# 9/ Close the ruleset

close(ruleset_fd);
```

0003-WORKSHOP-Create-a-ruleset.patch

Patch ImageMagick 5/9

```
# 10/

if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
    perror("LANDLOCK: Failed to lock privileges");
    return MagickFalse;
}

if (landlock_restrict_self(ruleset_fd, 0)) {
    perror("LANDLOCK: Failed to restrict thread");
    return MagickFalse;
}
```

0004-WORKSHOP-Restrict-and-break-everything.patch

Build and test the patched ImageMagick

```
# Regularly build and check convert

make && convert /vagrant/exploit/malicious.mvg /tmp/out.png
```

Patch ImageMagick 6/9

```
    # 11/ Add static rules: exceptions to the denied-by-default policy

+   struct landlock_path_beneath_attr rule;
+
+   printf("LANDLOCK: Adding rule for /usr");
+   rule.parent_fd = open("/usr", O_PATH | O_CLOEXEC);
+   rule.allowed_access = ACCESS_FS_ROUGHLY_READ;
+   if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0)) {
+       perror("LANDLOCK: Failed to create rule");
+       return MagickFalse;
+   }
+   close(rule.parent_fd);


    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
```

0005-WORKSHOP-Add-static-restrictions.patch

Patch ImageMagick 7/9

```
    # 12/ Add more static rules: /dev/null and /tmp (with appropriate access)

+   printf("LANDLOCK: Adding rule for /dev/null");
+   rule.parent_fd = open("/dev/null", O_PATH | O_CLOEXEC);
+   rule.allowed_access = LANDLOCK_ACCESS_FS_READ_FILE;
+   if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0)) {
+       perror("LANDLOCK: Failed to create rule");
+       return MagickFalse;
+   }
+   close(rule.parent_fd);


    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
```

0005-WORKSHOP-Add-static-restrictions.patch

Patch ImageMagick 8/9

```
    # 13/ Add a dynamic rule according to CLI arguments

+   printf("LANDLOCK: Adding rule for %s", filename);
+   rule.parent_fd = open(filename, O_PATH | O_CLOEXEC);
+   rule.allowed_access = LANDLOCK_ACCESS_FS_READ_FILE;
+   if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0)) {
+       perror("LANDLOCK: Failed to create rule");
+       return MagickFalse;
+   }
+   close(rule.parent_fd);


    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
```

0006-WORKSHOP-Handle-input-and-output-files.patch

Patch ImageMagick 9/9

```
    # 14/ Add more dynamic rules

+   char *out_path = strdup(argv[i+1]);
+   const char *out_dir = dirname(out_path);
+   [...]


    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0))
```

`0006-WORKSHOP-Handle-input-and-output-files.patch`

Build and test the final version

```
# Build and check convert

make && convert /vagrant/exploit/malicious.mvg /tmp/out.png
```

Exercise left to the readers

- Make the code more generic and maintainable
- Support the “fd:” URI scheme
- Support more commands
- Test with different kernel versions with help from the Landlock test tools

- …and send your patch upstream!

Compatibility and best-effort security

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.

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.

Good sandboxing rules

1. Transparent to users
2. Best-effort with minimal requirement
3. Handle strict restrictions
4. Runtime configuration with maximum execution

Rule #1: Transparent to users

Most of the time, configurations are not updated.

Requirements:

- Leverage the current application’s configuration as much as possible
- Dynamic checks to identify required runtime resources

Rule #2: Best-effort with minimal requirement

Don’t break my application!

Enforce **restrictions as much as possible** according to the running kernel, and being able to disable the whole sandboxing if a required feature is not supported (e.g., the refer access right for file reparenting).

Use case:

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

Rule #3: Handle strict restrictions

Create an option to force sandboxing and error out if anything goes wrong (not enabled by default).

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**

Rule #4: Runtime configuration with maximum execution

Help **identify sandboxing specific code issues**.

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

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

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

Wrap-up

ImageMagick patch

- Use the native CLI arguments:
  - Transparent for users
  - Well integrated with all supported use cases
- Quick to implement a first PoC
- Quicker when we already know the app code

Roadmap

Ongoing and next steps:

- Add new access-control types: socket creation, UDP port use…
- Add audit support to ease debugging and provide metrics
- Develop a new sandboxer tool
- Improve adoption

Contribute

- Develop new new access types
- Improve libraries: Rust, Go…
- Challenge the implementation
- Improve documentation or tests
- **Sandbox your applications** and others'
  - Secure Open Source Rewards
  - Google Patch Rewards

Questions?

landlock@lists.linux.dev

Thank you!