

Landlock workshop: Sandboxing application in practice

Let's sandbox ImageMagick!

Mickaël Salaün

Sandboxing an application

Landlock is available in mainline since 2021 (Linux 5.13), but with some limitations due to the iterative approach.

Landlock is now enabled by default on multiple distros: [Ubuntu 22.04 LTS](#), [Fedora 35](#), [Arch Linux](#), [Alpine Linux](#), Gentoo, Debian Sid, chromeOS, CBL-Mariner, WSL2

This workshop is about sandboxing ImageMagick

Workshop setup

Goal of this workshop

About the steps to sandbox a CLI application.

Use an old and vulnerable (long-been-fixed) ImageMagick version to illustrate how sandboxing can mitigate vulnerabilities.

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
3. Install the created package

Sandboxing with Landlock

Developers and users

It is assumed that with enough skills and time, most applications could be compromised.

Problem (as developers):

- We don't want to participate to malicious actions through our software because of security bug exploitation.
- We have a responsibility for users, especially to protect their (personal) data: every **running app/service increases (user) attack surface.**

What is Landlock?

Landlock is an access control system available to **unprivileged** processes on Linux, thanks to 3 dedicated syscalls.

It enables developers to add **built-in** application **sandboxing**.

Useful as-is and still in gaining new features.

Implementing sandboxing

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...

Step 1: Check the Landlock ABI

```
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 |
        [...]
        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 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");
```


Step 4: Enforce the ruleset

```
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);
```

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

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).

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

2/ Prepare a clean repository to work on

```
/vagrant/imagemagick-patches/init-repo.sh
```


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

```
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 prepare a ruleset

```
(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,  
+ };
```

Build and test the patched ImageMagick

```
# Regularly build and check convert
```

```
make
```

```
./utilities/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);
```

Patch ImageMagick 5/9

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

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

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

Build and test the patched ImageMagick

```
# Regularly build and check convert
```

```
make
```

```
./utilities/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.handled_access_fs = ACCESS_FS_ROUGHLY_READ;
+ if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0))
+     perror("LANDLOCK: Failed to create rule");

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

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.handled_access_fs = LANDLOCK_ACCESS_FS_READ_FILE;  
+ if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0))  
+     perror("LANDLOCK: Failed to create rule");  
+  
+  
+ [...]
```

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


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.handled_access_fs = LANDLOCK_ACCESS_FS_READ_FILE;
+ if (landlock_add_rule(ruleset_fd, LANDLOCK_RULE_PATH_BENEATH, &rule, 0))
+     perror("LANDLOCK: Failed to create rule");
+
+
+ [...]
```

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

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))
```

Build and install the patched ImageMagick

```
# Once everything looks OK, build and install the package
```

```
cd ../..
```

```
makepkg -efi --nocheck
```

```
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
- Build a new kernel with the KUnit Landlock patch and run tests
- Test with different kernel versions thanks to 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
- ...
- 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

Contribute

- Develop new (kernel) features (e.g., new access types)
 - See [GitHub issues: landlock-lsm/linux](#)
- Write new tests (Kselftest or KUnit)
- Challenge the implementation
- Improve documentation
- **Sandbox your applications** and others'
 - [Secure Open Source Rewards](#)
 - [Google Patch Rewards](#)

Questions?

<https://docs.kernel.org/userspace-api/landlock.html>

Past talks: <https://landlock.io>

landlock@lists.linux.dev

Thank you!