How to safely restrict access to files in a programmatic way with Landlock?

Mickaël Salaün

ANSSI

August 27, 2018
Part 1: Why Landlock, what is it and how does it work? (quick recap)
Designed to create tailored security sandboxes

**Threat**

bug exploitation or backdoor in an application (client or server side)
Designed to create tailored security sandboxes

**Threat**
bug exploitation or backdoor in an application (client or server side)

**Goal**
protect user of the application against unintended accesses
Features and use cases

Tailored security policy, by the developer

▶ e.g. able to choose the security model that fit best
▶ e.g. embedded in an application and evolve with it
▶ e.g. use application’s configuration
Features and use cases

Tailored security policy, by the developer

▶ e.g. able to choose the security model that fit best
▶ e.g. embedded in an application and evolve with it
▶ e.g. use application’s configuration

Compose access controls from multiple tenants

▶ e.g. sysadmin, end user and developers
▶ e.g. multiple cloud clients
Features and use cases

Tailored security policy, by the developer

▶ e.g. able to choose the security model that fit best
▶ e.g. embedded in an application and evolve with it
▶ e.g. use application’s configuration

Compose access controls from multiple tenants

▶ e.g. sysadmin, end user and developers
▶ e.g. multiple cloud clients

Able to update access control on the fly

▶ e.g. native powerbox support (file picker, portal...)
▶ e.g. dynamic policy update according to external factors
Demonstration #1

Read-only accesses...

► /public
► /etc
► /usr
► ...

...and read-write accesses

► /tmp
► ...

5 / 19
Landlock overview

process

user-space

call open(...)

restrict

Landlock programs

kernel-space

open
Gears of Landlock

Linux Security Modules (LSM)

- allow or deny user-space actions on kernel objects
- 200+ hooks: inode_permission, inode_unlink, file_ioctl...
Gears of Landlock

Linux Security Modules (LSM)

- allow or deny user-space actions on kernel objects
- 200+ hooks: inode_permission, inode_unlink, file_ioctl...

extended Berkeley Packet Filter (eBPF)

- safely interpret bytecode in the kernel at run time
- can call dedicated functions
- can exchange data through maps between eBPF programs and user-space
Gears of Landlock

Linux Security Modules (LSM)

- allow or deny user-space actions on kernel objects
- 200+ hooks: inode_permission, inode_unlink, file_ioctl...

extended Berkeley Packet Filter (eBPF)

- safely interpret bytecode in the kernel at run time
- can call dedicated functions
- can exchange data through maps between eBPF programs and user-space

Landlock

- hook: set of actions on a specific kernel object
- program: access control checks stacked on a hook
- triggers: actions mask for which a program is run
Unprivileged access control

Protect access to process resources

- the process requesting to apply a new access control must be allowed to `ptrace` the sandboxed process
Unprivileged access control

Protect access to process resources

- the process requesting to apply a new access control must be allowed to ptrace the sandboxed process

Protect access to kernel resources

- prevent information leak: an eBPF program shall not have access to informations not otherwise granted to the process requesting the sandboxing
- avoid side-channels: only interpreted on viewable objects and after other access controls
- account kernel resources used by the access controls
Part 2: Why and how the filesystem access control is different between Landlock and other LSMs?
Inode’s extended attributes (xattr)

Pros

▶ native and efficient for the kernel to identify a file access
Inode’s extended attributes (xattr)

Pros
▶ native and efficient for the kernel to identify a file access

Cons (for Landlock)
▶ no composability: only one label/view per inode (hard link, bind mounts, namespaces...)  
▶ not unprivileged:  
  ▶ no (efficient) accounting per access control  
  ▶ need a filesystem which support xattr  
  ▶ need write access to label a file  
▶ not dynamic: impose a persistent labelling
Pros

▶ point of view of the user
Pros

▶ point of view of the user

Cons (for Landlock)

▶ composability: need to remember how a file was (relatively) accessed
▶ unprivileged:
  ▶ dealing with underlying inode can be tricky: partial path, anonymous inodes, chroot, namespaces. . .
  ▶ risk of leaking path informations
A new eBPF map type to identify an inode

- filled with a reference to the inode pointed by a file descriptor
- efficient inode matching
- updatable from user-space
- unprivileged use
eBPF inode map

A new eBPF map type to identify an inode

- filled with a reference to the inode pointed by a file descriptor
- efficient inode matching
- updatable from user-space
- unprivileged use

Properties

- inode identification not stored on the filesystem but (accounted) in the map
- use inode as key and associate it with a 64-bits arbitrary value
Demonstration #2

Update access rights on the fly
Chained programs and session

Landlock programs and their triggers (example)

fs_walk
Chained programs and session

Landlock programs and their triggers (example)

```
fs_walk

fs_pick: open, chdir, getattr...
```
Chained programs and session

Landlock programs and their triggers (example)

fs_walk

fs_pick    open, chdir, getattr...

fs_pick    create, write, link...
Walking through a file path

Example: open /public/web/index.html

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>1</td>
</tr>
<tr>
<td>/public</td>
<td>1</td>
</tr>
<tr>
<td>/tmp</td>
<td>1</td>
</tr>
</tbody>
</table>
Walking through a file path

Example: open /public/web/index.html

cookie = 0

/  

ds_walk #1

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>1</td>
</tr>
<tr>
<td>/public</td>
<td>1</td>
</tr>
<tr>
<td>/tmp</td>
<td>1</td>
</tr>
</tbody>
</table>
Walking through a file path

Example: open /public/web/index.html

```
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>1</td>
</tr>
<tr>
<td>/public</td>
<td>1</td>
</tr>
<tr>
<td>/tmp</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Walking through a file path

Example: open /public/web/index.html
Walking through a file path

Example: open /public/web/index.html

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>1</td>
</tr>
<tr>
<td>/public</td>
<td>1</td>
</tr>
<tr>
<td>/tmp</td>
<td>1</td>
</tr>
</tbody>
</table>
Walking through a file path

Example: open /public/web/index.html

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>1</td>
</tr>
<tr>
<td>/public</td>
<td>1</td>
</tr>
<tr>
<td>/tmp</td>
<td>1</td>
</tr>
</tbody>
</table>
Identifying access to a subset of the filesystem, the Landlock way

Pros

▶ agnostic to chroot and namespaces
▶ no need for extra informations (not already available to the requester process)
▶ accountable security policy
▶ updatable on the fly
▶ do not rely on string matching
▶ can still rely on file hierarchy... this way or another
▶ easy to implement tests

Cons
Identifying access to a subset of the filesystem, the Landlock way

Pros

▶ agnostic to chroot and namespaces
▶ no need for extra informations (not already available to the requester process)
▶ accountable security policy
▶ updatable on the fly
▶ do not rely on string matching
▶ can still rely on file hierarchy... this way or another
▶ easy to implement tests

Cons

▶ rely on the way the kernel does (relative) pathname lookup (e.g. symlinks, dot, dotdot)
▶ add a security blob to nameidata
Identifying access to a subset of the filesystem, the Landlock way

Concern from the filesystem kernel developers might rely too much on the current pathname lookup implementation, which changed multiple times until 2000 (cf. header comments in fs/namei.c)
Identifying access to a subset of the filesystem, the Landlock way

**Concern from the filesystem kernel developers**

might rely too much on the current pathname lookup implementation, which changed multiple times until 2000 (cf. header comments in fs/namei.c)

**However...**

- this logic is already visible and used by DAC and MAC systems
- ...and user-defined policies
Landlock: wrap-up

User-space hardening

- programmatic and embeddable access control
- designed for unprivileged use

Current status

- security/landlock/*:
  - ∼2000 SLOC
- ongoing patch series: LKML, @l0kod
- figuring out about the pathname lookup concerns
- full security module stacking is coming!

Further along the way

- audit support
- extend access control: network, IPC, ...
- (real) (programmable) capabilities
- library and tools
Landlock: wrap-up

User-space hardening

- programmatic and embeddable access control
- designed for unprivileged use

Current status

- security/landlock/*: ~2000 SLOC
- ongoing patch series: LKML, @l0kod
- figuring out about the pathname lookup concerns
- full security module stacking is coming!
Landlock: wrap-up

User-space hardening

► programmatic and embeddable access control
► designed for unprivileged use

Current status

► security/landlock/*: ~2000 SLOC
► ongoing patch series: LKML, @l0kod
► figuring out about the pathname lookup concerns
► full security module stacking is coming!

Further along the way

► audit support
► extend access control: network, IPC...
► (real) (programmable) capabilities
► library and tools
Life cycle of a Landlock program

C source → eBPF bytecode → application → process → kernel

- build program
- embed program
- execute application
- load program
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \ 
                    LANDLOCK_TRIGGER_FS_PICK_CREATE | \ 
                    // [...] 
                    LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \  
                  LANDLOCK_TRIGGER_FS_PICK_CREATE | \  
                  // [...]
        LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
Landlock program’s metadata

```c
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \
                    LANDLOCK_TRIGGER_FS_PICK_CREATE | \
                    // [...]
                    LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
```
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \
            LANDLOCK_TRIGGER_FS_PICK_CREATE | \
                // [...] \n            LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
```
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \
                   LANDLOCK_TRIGGER_FS_PICK_CREATE | \ 
                    // [...] \
                   LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
```
Landlock program’s metadata

```c
static union bpf_prog_subtype metadata = {
    .landlock_hook = {
        .type = LANDLOCK_HOOK_FS_PICK,
        .options = LANDLOCK_OPTION_PREVIOUS,
        .previous = 2, /* landlock2 */
        .triggers = LANDLOCK_TRIGGER_FS_PICK_APPEND | \    
                     LANDLOCK_TRIGGER_FS_PICK_CREATE | \    
                     // [...] \    
                      LANDLOCK_TRIGGER_FS_PICK_WRITE,
    }
};
```
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup, (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;

    return LANDLOCK_RET_DENY;
}
```c
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup,
                            (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;

    return LANDLOCK_RET_DENY;
}
```
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup,
                           (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;
    return LANDLOCK_RET_DENY;
}
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup, (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;
    return LANDLOCK_RET_DENY;
}
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup,
                             (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;

    return LANDLOCK_RET_DENY;
}
```c
int fs_pick_write(struct landlock_ctx_fs_pick *ctx) {
    __u64 cookie = ctx->cookie;

    cookie = update_cookie(cookie, ctx->inode_lookup, (void *)ctx->inode);

    if (cookie & MAP_MARK_WRITE)
        return LANDLOCK_RET_ALLOW;

    return LANDLOCK_RET_DENY;
}
```
Loading a rule in the kernel

```c
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK,
    .prog_subtype = &metadata,
    // [...]
};

int prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK,
    .prog_subtype = &metadata,
    // [...]
};

int prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK,
    .prog_subtype = &metadata,
    // [...]
};

int prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
union bpf_attr attr = {
  .insns = bytecode_array,
  .prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK,
  .prog_subtype = &metadata,
  // [...]
};

int prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK,
    .prog_subtype = &metadata,
    // [...]
};
int prog_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

![Diagram showing process calling bpf(...) which loads a program in kernel-space, with user-space separated by a dashed line.](image)
Applying a Landlock program to a process

```c
1 | seccomp(SECCOMP_PREPEND_LANDLOCK_PROG, 0, &prog_fd);
```
Applying a Landlock program to a process
Applying a Landlock program to a process
Applying a Landlock program to a process

```
process
  open(...)
  LSM hook
    Landlock hook
    program
  open
kernel-space
```

user-space
Kernel execution flow

Example: the `inode_create` hook

1. check if `landlocked(current)`
2. call `decide_fs_pick(LANDLOCK_TRIGGER_FS_PICK_CREATE, dir)`
3. for all `fs_pick` programs enforced on the current process
   3.1 update the program's context
   3.2 interpret the program
   3.3 continue until one denies the access
Rule enforcement on process hierarchy

P1
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy

Diagram:
- P1
- P2
- P3
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy
Enforcement through cgroups

Why?
user/admin security policy (e.g. container): manage groups of processes
Enforcement through cgroups

Why?
user/admin security policy (e.g. container): manage groups of processes

Challenges
▶ complementary to the process hierarchy rules (via seccomp(2))
▶ processes moving in or out of a cgroup
▶ unprivileged use with cgroups delegation (e.g. user session)
Future Landlock program types

\texttt{fs\_get}

tag inodes: needed for relative path checks (e.g. \textit{openat(2)})
Future Landlock program types

fs_get

tag inodes: needed for relative path checks (e.g. openat(2))

fs_ioctl

tag IOCTL commands
Future Landlock program types

**fs_get**
tag inodes: needed for relative path checks (e.g. `openat(2)`)  

**fs_ioctl**
check IOCTL commands  

**net_***
check IPs, ports, protocol...