Landlock LSM: toward unprivileged sandboxing

Mickaël SALAÜN

ANSSI

September 14, 2017
Secure user-space software

How to harden an application?

- secure development
- follow the least privilege principle
- compartmentalize exposed processes
Secure user-space software

How to harden an application?

▶ secure development
▶ follow the least privilege principle
▶ compartmentalize exposed processes

Multiple sandbox uses

▶ built-in sandboxing (tailored security policy)
▶ sandbox managers (unprivileged and dynamic compartmentalization)
▶ container managers (hardened containers)
What can provide the needed features?

<table>
<thead>
<tr>
<th>Fine-grained control</th>
<th>Embedded policy</th>
<th>Unprivileged use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELinux...</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
What can provide the needed features?

<table>
<thead>
<tr>
<th></th>
<th>Fine-grained control</th>
<th>Embedded policy</th>
<th>Unprivileged use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELinux...</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seccomp-bpf</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>namespaces</td>
<td></td>
<td>✓</td>
<td>~</td>
</tr>
</tbody>
</table>
What can provide the needed features?

<table>
<thead>
<tr>
<th></th>
<th>Fine-grained control</th>
<th>Embedded policy</th>
<th>Unprivileged use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELinux…</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seccomp-bpf</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>namespaces</td>
<td>✓</td>
<td></td>
<td>~</td>
</tr>
<tr>
<td>Landlock</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Tailored access control to match your needs: programmatic access control
What can provide the needed features?

<table>
<thead>
<tr>
<th></th>
<th>Fine-grained control</th>
<th>Embedded policy</th>
<th>Unprivileged use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELinux...</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seccomp-bpf</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>namespaces</td>
<td>✓</td>
<td></td>
<td>~</td>
</tr>
<tr>
<td>Landlock</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Tailored access control to match your needs: programmatic access control

Example
Run an application allowed to write only on a terminal.
Landlock overview
Landlock: patch v7

- a minimum viable product
- a stackable LSM
- using eBPF
- focused on filesystem access control
The Linux Security Modules framework (LSM)

LSM framework

- allow or deny user-space actions on kernel objects
- policy decision and enforcement points
- kernel API: support various security models
- 200+ hooks: inode_permission, inode_unlink, file_ioctl...
The Linux Security Modules framework (LSM)

**LSM framework**
- allow or deny user-space actions on kernel objects
- policy decision and enforcement points
- kernel API: support various security models
- 200+ hooks: `inode_permission`, `inode_unlink`, `file_ioctl`...

**Landlock**
- rule: control an action on an object
- event: use of a kernel object type (e.g. file)
- action: read, write, execute, remove, IOCTL...
Life cycle of a Landlock rule

C source

⇒

eBPF bytecode

⇒

application

⇒

process

⇒

kernel

build rule
embed rule
execute application
load rule
Landlock rule example

- read-only access to the filesystem...
- ...but allowed to write on TTY and pipes
- rule enforced on each filesystem access request
Landlock rule example

```
SEC("landlock1")
int landlock_fs_rule1(struct landlock_context *ctx)
{
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);

    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")
int landlock_fs_rule1(struct landlock_context *ctx) {
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);

    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")

int landlock_fs_rule1(struct landlock_context *ctx) {
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);

    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")
int landlock_fs_rule1(struct landlock_context *ctx)
{
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
    return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);
    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;
    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")
int landlock_fs_rule1(struct landlock_context *ctx)
{
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);

    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
```c
SEC("landlock1")

int landlock_fs_rule1(struct landlock_context *ctx)
{
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    /* allow write on TTY and pipes */
    mode = bpf_handle_fs_get_mode(ctx->arg1);
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")

int landlock_fs_rule1(struct landlock_context *ctx) {

    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    mode = bpf_handle_fs_get_mode(ctx->arg1);

    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
Landlock rule example

```c
SEC("landlock1")
int landlock_fs_rule1(struct landlock_context *ctx)
{
    int mode;

    /* allow non-write actions */
    if (!(ctx->arg2 & LANDLOCK_ACTION_FS_WRITE))
        return 0;

    /* get the file mode */
    /* allow write on TTY and pipes */
    if (S_ISCHR(mode) || S_ISFIFO(mode))
        return 0;

    return 1;
}
```
extended Berkeley Packet Filter

In-kernel virtual machine

- safely execute code in the kernel at run time
- widely used in the kernel: network filtering, seccomp-bpf, tracing...
- can call dedicated functions
- can exchange data through maps between eBPF programs and user-space
extended Berkeley Packet Filter

In-kernel virtual machine

- safely execute code in the kernel at run time
- widely used in the kernel: network filtering, seccomp-bpf, tracing...
- can call dedicated functions
- can exchange data through maps between eBPF programs and user-space

Static program verification at load time

- memory access checks
- register typing and tainting
- pointer leak restrictions
- execution flow restrictions
Loading a rule in the kernel

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};

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

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

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_RULE,
    .prog_subtype = &metadata,
    // [...]
};
int rule_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_RULE,
    .prog_subtype = &metadata,
    // [...]
};
int rule_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};
union bpf_attr attr = {
    .insns = bytecode_array,
    .prog_type = BPF_PROG_TYPE_LANDLOCK_RULE,
    .prog_subtype = &metadata,
    // [...]
};
int rule_fd = bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```
Loading a rule in the kernel

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};

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

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

```c
static union bpf_prog_subtype metadata = {
    .landlock_rule = {
        .event = LANDLOCK_EVENT_FS,
        .ability = LANDLOCK_ABILITY_DEBUG,
    }
};

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

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

process

rule

user-space

kernel-space

bpf(...)

10 / 21
Applying a rule to a process

```c
1 | seccomp(SECCOMP_PREPEND_LANDLOCK_RULE, 0, &rule_fd);
```
Applying a rule to a process

Diagram:
- Process output
- Bpf(...) function
- Rule
- User-space
- Kernel-space
Applying a rule to a process

- **process**
- **seccomp(...)**
- **rule**
- **user-space**
- **kernel-space**
Applying a rule to a process

- process
- write(...)
- hook
- event
- write
- rule

user-space

kernel-space
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy
Rule enforcement on process hierarchy

Diagram:
- P1
- P2
- P3
Rule enforcement on process hierarchy
Demonstration #1

An (almost) read-only filesystem
Landlock: pending features

- unprivileged access control
- enforcement through cgroups
- eBPF map fsview
- coming next...
Unprivileged access control

Why?
embed a security policy in any application, following the least privilege principle
Unprivileged access control

Why?
embed a security policy in any application, following the least privilege principle

Challenges
- applying a security policy requires privileges
- unlike SUID, Landlock should only reduce accesses
- prevent accesses through other processes: ptrace restrictions
- protect the kernel: eBPF static analysis
- prevent information leak: an eBPF program shall not have more access rights than the process which loaded it
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)
eBPF map fsview

Why?
restrict access to a subset of the filesystem
eBPF map fsview

Why?
restrict access to a subset of the filesystem

Challenges

▶ efficient
▶ updatable from user-space
▶ unprivileged use (i.e. no xattr)
eBPF map fsview

Why?
restrict access to a subset of the filesystem

Challenges
▶ efficient
▶ updatable from user-space
▶ unprivileged use (i.e. no xattr)

Proposal
▶ new eBPF map to identify a *filesystem view*: mount point hierarchies at a given time
▶ new eBPF function to compare a file access to such a view
Demonstration #2

What might a filesystem access control looks like?
Current roadmap

Incremental upstream integration

1. minimum viable product
2. cgroups handling
3. new eBPF map type for filesystem-related checks
4. unprivileged mode
Landlock: wrap-up

User-space hardening

- programmatic access control
- designed for unprivileged use
Landlock: wrap-up

User-space hardening

- programmatic access control
- designed for unprivileged use

Current status: patch v7

- autonomous patches merged in net, security and kselftest trees
- security/landlock/*/: ~1K SLOC
- ongoing patch series: LKML, @l0kod
- growing interest for containers, secure OS and service managers
Landlock context

```c
struct landlock_context {
    __u64 status;
    __u64 event;
    __u64 arg1;
    __u64 arg2;
};
```
Landlock context

```c
struct landlock_context {
    __u64 status;
    __u64 event;
    __u64 arg1;
    __u64 arg2;
}
```

Landlock events

- LANDLOCK_EVENT_FS
Landlock context

```c
struct landlock_context {
    __u64 status;
    __u64 event;
    __u64 arg1;
    __u64 arg2;
};
```

Landlock events

- LANDLOCK_EVENT_FS
Landlock context

```c
struct landlock_context {
    __u64 status;
    __u64 event;
    __u64 arg1;
    __u64 arg2;
};
```

Landlock events

- LANDLOCK_EVENT_FS

Landlock actions for an FS event

- LANDLOCK_ACTION_FS_EXEC
- LANDLOCK_ACTION_FS_WRITE
- LANDLOCK_ACTION_FS_READ
- LANDLOCK_ACTION_FS_NEW
- LANDLOCK_ACTION_FS_GET
- LANDLOCK_ACTION_FS_REMOVE
- LANDLOCK_ACTION_FS_IOCTL
- LANDLOCK_ACTION_FS_LOCK
- LANDLOCK_ACTION_FS_FCNTRL
Landlock context

```
struct landlock_context {
    __u64 status;
    __u64 event;
    __u64 arg1;
    __u64 arg2;
};
```

Landlock events

- LANDLOCK_EVENT_FS
- LANDLOCK_EVENT_FS_IOCTL
- LANDLOCK_EVENT_FS_LOCK
- LANDLOCK_EVENT_FS_FCNTL

Landlock actions for an FS event

- LANDLOCK_ACTION_FS_EXEC
- LANDLOCK_ACTION_FS_WRITE
- LANDLOCK_ACTION_FS_READ
- LANDLOCK_ACTION_FS_NEW
- LANDLOCK_ACTION_FS_GET
- LANDLOCK_ACTION_FS_REMOVE
- LANDLOCK_ACTION_FS_IOCTL
- LANDLOCK_ACTION_FS_LOCK
- LANDLOCK_ACTION_FS_FCNTL
Available eBPF functions for Landlock rules

Any rule

- bpf_handle_fs_get_mode
Available eBPF functions for Landlock rules

Any rule
  ▶ bpf_handle_fs_get_mode

Debug mode: need CAP_SYS_ADMIN
  ▶ bpf_get_current_comm
  ▶ bpf_get_current_pid_tgid
  ▶ bpf_get_current_uid_gid
  ▶ bpf_get_trace_printk