Landlock: a new kind of Linux Security Module leveraging eBPF

Mickaël Salaün

ANSSI

September 12, 2019

Protect users from your application

Threat

1. bug exploitation of your code
2. bug or backdoor in a third party component

$\Rightarrow$ your application is used against your will

Protect users from your application

Threat

1. bug exploitation of your code
2. bug or backdoor in a third party component

$\Rightarrow$ your application is used against your will

Defenses

- follow secure development practices
- use an hardened toolchain
- use OS security features (e.g. sandboxes)

Protect users from your application

Threat

1. bug exploitation of your code
2. bug or backdoor in a third party component

$\Rightarrow$ your application is used against your will

Defenses

- follow secure development practices
- use an hardened toolchain
- use OS security features (e.g. sandboxes)

The Landlock features (current)

- helps define and embed security policy in your code
- enforces an access control on your application

Demonstration #1 [PATCH v8]

Read-only accesses...

- `/public`
- `/etc`
- `/usr`
- `...`

...and read-write accesses

- `/tmp`
- `...`

What about the other Linux security features?

	Fine-grained control	**Embedded policy**	**Unprivileged use**
SELinux...	✓		

What about the other Linux security features?

	Fine-grained control	**Embedded policy**	**Unprivileged use**
SELinux...	✓		
seccomp-bpf		✓	✓
namespaces		✓	$\sim$

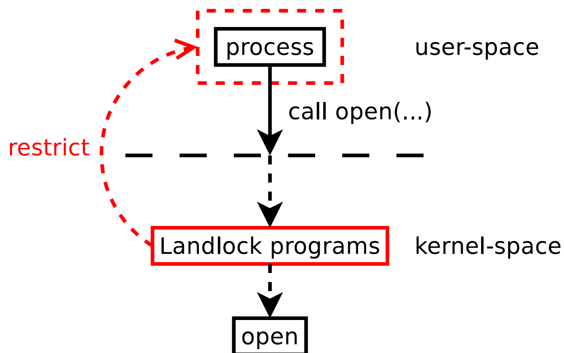
What about the other Linux security features?

	Fine-grained control	Embedded policy	Unprivileged use
SELinux...	✓		
seccomp-bpf		✓	✓
namespaces		✓	∼
Landlock	✓	✓	✓[1]

Tailored access control to match your needs: programmatic access control

[1] Disabled on purpose for the initial upstream inclusion, but planned to be enabled after a test period (and subject to upstream point of view).

Landlock overview



extended Berkeley Packet Filter

In-kernel virtual machine

- safely execute code in the kernel at run time
- widely used in the kernel: network filtering (XDP), 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 (XDP), 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

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

- hook: set of actions on a specific kernel object (e.g. walk a file path, change memory protection)
- program: access-control checks stacked on a hook
- triggers: actions mask for which a program is run (e.g. read, write, execute, remove, IOCTL...)

History of Landlock

Overview of the major patch series

- [PATCH v1] (Mar. 2016): seccomp-object
- [PATCH v2] (Aug. 2016): LSM + cgroups
- [PATCH v8] (Feb. 2018): file path identification
- [PATCH v10] (Jul. 2019): shrink patches (current version)

Safely handle malicious policies

- Landlock should be usable by everyone
- we can't tell if a process will be malicious

$\Rightarrow$ trust issue

Unprivileged access control

Sought properties

- multiple applications, need independant but composable security policies
- tamper proof: prevent bypass through other processes (i.e. via ptrace)

Unprivileged access control

Sought properties

- multiple applications, need independant but composable security policies
- tamper proof: prevent bypass through other processes (i.e. via ptrace)

Harmlessness

- safe approach: follow the least privilege principle (i.e. no SUID)
- limit the kernel attack surface:
  - minimal kernel code (`security/landlock/*`: $\sim$1080 SLOC)
  - eBPF static analysis
  - move complexity from the kernel to eBPF programs

Unprivileged access control

Protect access to process ressources

- the rule creator must be allowed to ptrace the sandboxed process

Unprivileged access control

Protect access to process ressources

- the rule creator must be allowed to ptrace the sandboxed process

Protect access to kernel ressources

- prevent information leak: an eBPF program shall not have more access rights than the process which loaded it
- still, access control need some knowledge to take decision (e.g. file path check)
- only interpreted on viewable objects and after other access controls

Identifying a file path

- path evaluation based on walking through inodes
- multiple Landlock program types

eBPF inode map

Goal

restrict access to a subset of the filesystem

eBPF inode map

Goal

restrict access to a subset of the filesystem

Challenges

- efficient
- updatable from user-space
- unprivileged use:
  - no xattr
  - no absolute path

eBPF inode map

Goal

restrict access to a subset of the filesystem

Challenges

- efficient
- updatable from user-space
- unprivileged use:
  - no xattr
  - no absolute path

Solution

- new eBPF map type to identify an inode object
- use inode as key and associate it with an arbitrary value

Demonstration #2 [PATCH v8]

Update access rights on the fly

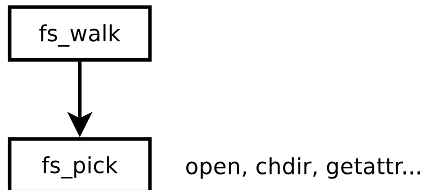
Chained programs and session [PATCH v8]

Landlock programs and their triggers (example)

fs_walk

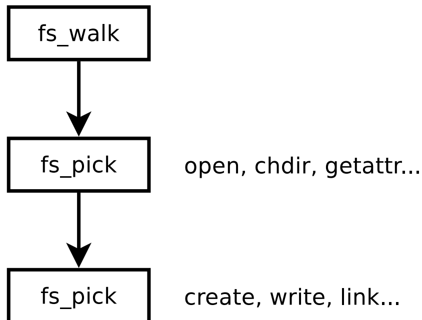
Chained programs and session [PATCH v8]

Landlock programs and their triggers (example)



Chained programs and session [PATCH v8]

Landlock programs and their triggers (example)



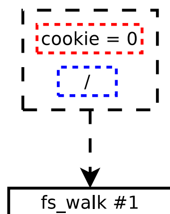
Walking through a file path [PATCH v8]

Example: `open /public/web/index.html`

key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

Walking through a file path [PATCH v8]

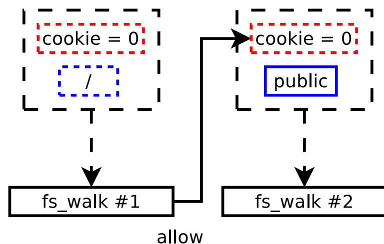
Example: `open /public/web/index.html`



key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

Walking through a file path [PATCH v8]

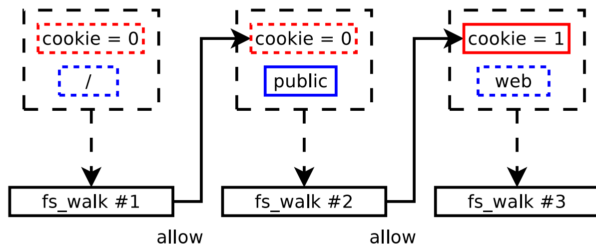
Example: `open /public/web/index.html`



key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

Walking through a file path [PATCH v8]

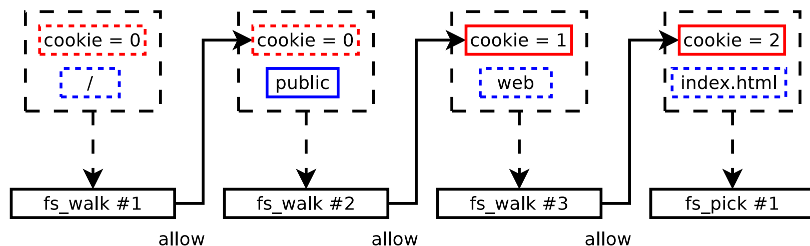
Example: `open /public/web/index.html`



key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

Walking through a file path [PATCH v8]

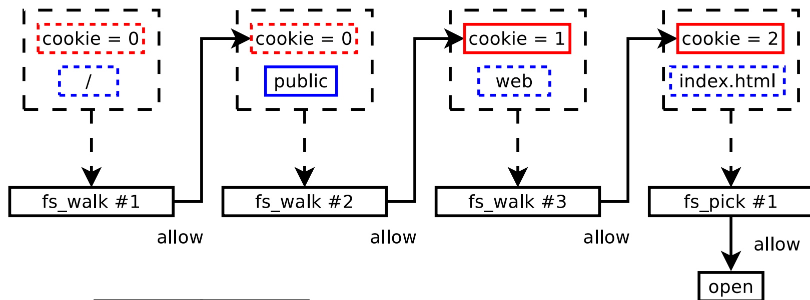
Example: `open /public/web/index.html`



key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

Walking through a file path [PATCH v8]

Example: `open /public/web/index.html`



key	value
/etc	1 (ro)
/public	1 (ro)
/tmp	2 (rw)

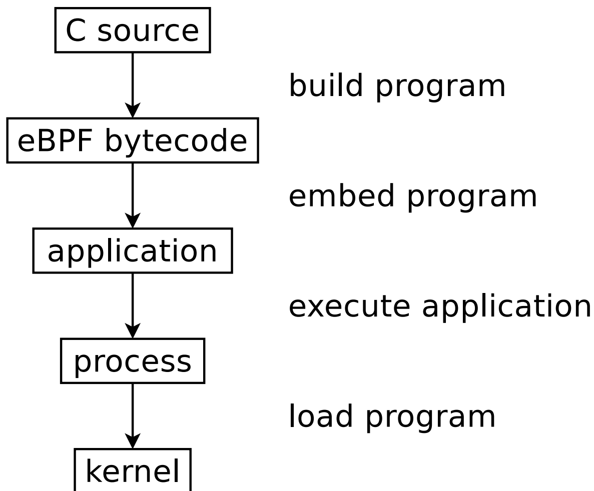
Demonstration #3 [PATCH v10]

- deny access to `~/.ssh` and `~/.gnupg`
- ptrace restriction

From the rule to the kernel

- writing a Landlock rule
- loading it in the kernel
- enforcing it on a set of processes

Life cycle of a Landlock program



Landlock eBPF inode map

```
1  struct bpf_map_def SEC("maps") inode_map = {
2      .type = BPF_MAP_TYPE_INODE,
3      .key_size = sizeof(u32),
4      .value_size = sizeof(u64),
5      .max_entries = MAP_MAX_ENTRIES,
6      .map_flags = BPF_F_RDONLY_PROG,
7  };
```

Landlock eBPF inode map

```
1 struct bpf_map_def SEC("maps") inode_map = {
2     .type = BPF_MAP_TYPE_INODE,
3     .key_size = sizeof(u32),
4     .value_size = sizeof(u64),
5     .max_entries = MAP_MAX_ENTRIES,
6     .map_flags = BPF_F_RDONLY_PROG,
7 };
```

Landlock eBPF inode map

```
struct bpf_map_def SEC("maps") inode_map = {
    .type = BPF_MAP_TYPE_INODE,
    .key_size = sizeof(u32),
    .value_size = sizeof(u64),
    .max_entries = MAP_MAX_ENTRIES,
    .map_flags = BPF_F_RDONLY_PROG,
};
```

Landlock eBPF inode map

```
1  struct bpf_map_def SEC("maps") inode_map = {
2      .type = BPF_MAP_TYPE_INODE,
3      .key_size = sizeof(u32),
4      .value_size = sizeof(u64),
5      .max_entries = MAP_MAX_ENTRIES,
6      .map_flags = BPF_F_RDONLY_PROG,
7  };
```

Landlock eBPF program code

```
1  SEC("landlock/fs_pick")
2  int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
3  {
4      u64 *flags;
5
6      flags = bpf_inode_map_lookup_elem(&inode_map,
7                                        (void *)ctx->inode);
8      if (flags && (*flags & MAP_FLAG_DENY))
9          return LANDLOCK_RET_DENY;
10     return LANDLOCK_RET_ALLOW;
11 }
```

Landlock eBPF program code

```
1  SEC("landlock/fs_pick")
2  int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
3  {
4      u64 *flags;
5  
6      flags = bpf_inode_map_lookup_elem(&inode_map,
7                                        (void *)ctx->inode);
8      if (flags && (*flags & MAP_FLAG_DENY))
9          return LANDLOCK_RET_DENY;
10     return LANDLOCK_RET_ALLOW;
11 }
```

Landlock eBPF program code

```
SEC("landlock/fs_pick")
int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
{
    u64 *flags;

    flags = bpf_inode_map_lookup_elem(&inode_map,
                                       (void *)ctx->inode);
    if (flags && (*flags & MAP_FLAG_DENY))
        return LANDLOCK_RET_DENY;
    return LANDLOCK_RET_ALLOW;
}
```

Landlock eBPF program code

```
 1  SEC("landlock/fs_pick")
 2  int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
 3  {
 4      u64 *flags;
 5
 6      flags = bpf_inode_map_lookup_elem(&inode_map,
 7                                        (void *)ctx->inode);
 8      if (flags && (*flags & MAP_FLAG_DENY))
 9          return LANDLOCK_RET_DENY;
10      return LANDLOCK_RET_ALLOW;
11  }
```

Landlock eBPF program code

```
1  SEC("landlock/fs_pick")
2  int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
3  {
4      u64 *flags;
5
6      flags = bpf_inode_map_lookup_elem(&inode_map,
7                                         (void *)ctx->inode);
8      if (flags && (*flags & MAP_FLAG_DENY))
9          return LANDLOCK_RET_DENY;
10     return LANDLOCK_RET_ALLOW;
11 }
```

Landlock eBPF program code

```
1   SEC("landlock/fs_pick")
2   int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
3   {
4       u64 *flags;
5
6       flags = bpf_inode_map_lookup_elem(&inode_map,
7                                         (void *)ctx->inode);
8       if (flags && (*flags & MAP_FLAG_DENY))
9           return LANDLOCK_RET_DENY;
10      return LANDLOCK_RET_ALLOW;
11  }
```

Landlock eBPF program code

```
1   SEC("landlock/fs_pick")
2   int fs_pick_ro(struct landlock_ctx_fs_pick *ctx)
3   {
4       u64 *flags;
5
6       flags = bpf_inode_map_lookup_elem(&inode_map,
7                                         (void *)ctx->inode);
8       if (flags && (*flags & MAP_FLAG_DENY))
9           return LANDLOCK_RET_DENY;
10      return LANDLOCK_RET_ALLOW;
11  }
```

Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

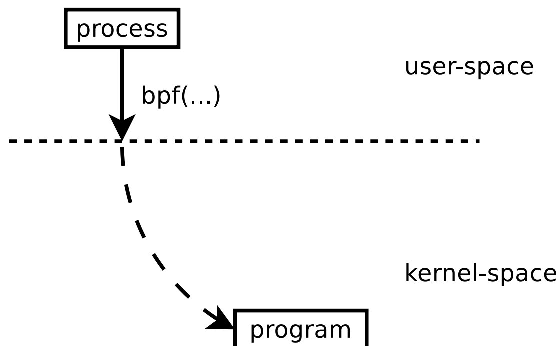
Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

Landlock program's metadata

```
1  struct bpf_load_program_attr load_attr = {};
2
3  load_attr.prog_type = BPF_PROG_TYPE_LANDLOCK_HOOK;
4  load_attr.expected_attach_type = BPF_LANDLOCK_FS_PICK;
5  load_attr.expected_attach_triggers = LANDLOCK_TRIGGER_FS_PICK_OPEN;
6  load_attr.insns = insns;
7  load_attr.insns_cnt = sizeof(insn) / sizeof(struct bpf_insn);
8  load_attr.license = "GPL";
9
10 int prog_fd = bpf_load_program_xattr(&load_attr, log_buf, log_buf_sz)
11 if (prog_fd == -1)
12     exit(1);
```

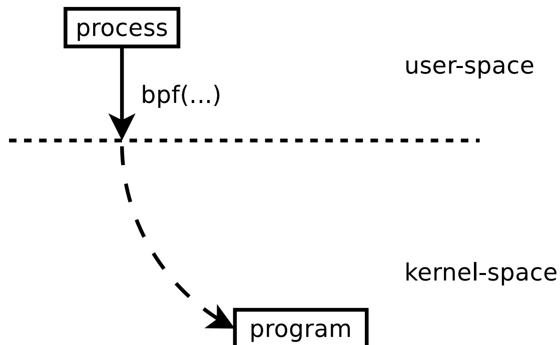
Loading a rule in the kernel



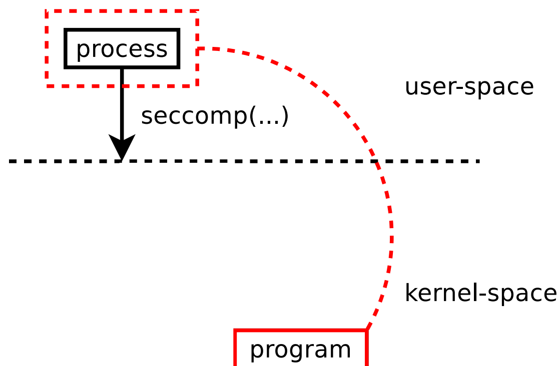
Applying a Landlock program to a process

```
1 | seccomp(SECCOMP_PREPEND_LANDLOCK_PROG, 0, &prog_fd);
```

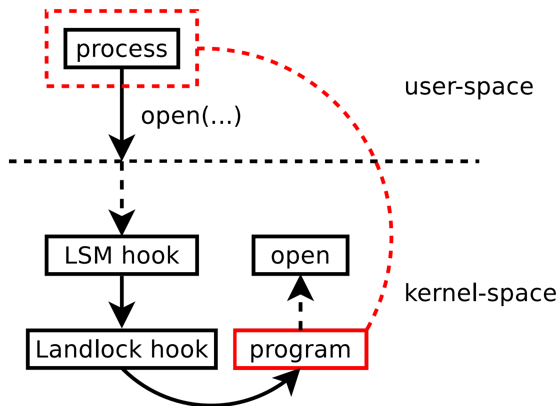
Applying a Landlock program to a process



Applying a Landlock program to a process



Applying a Landlock program to a process



Kernel execution flow

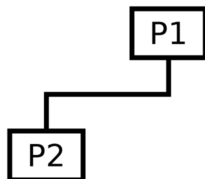
Example: the `inode_create` hook

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

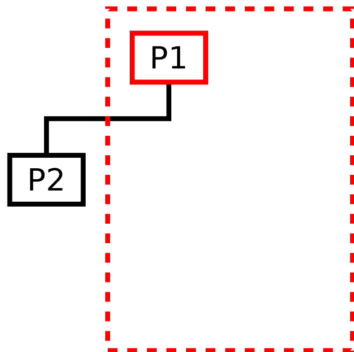
Rule enforcement on process hierarchy

P1

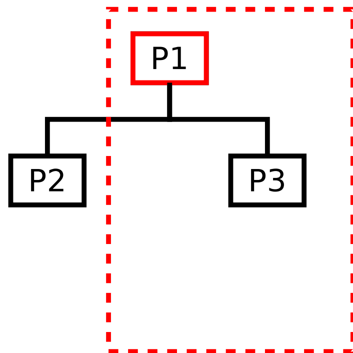
Rule enforcement on process hierarchy



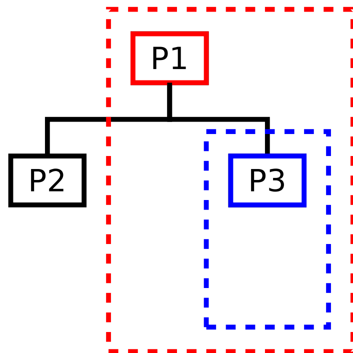
Rule enforcement on process hierarchy



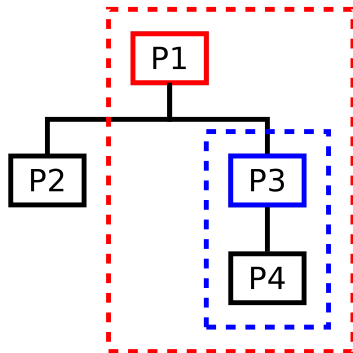
Rule enforcement on process hierarchy



Rule enforcement on process hierarchy



Rule enforcement on process hierarchy



Enforcement through cgroups [PATCH v4]

Why?

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

Enforcement through cgroups [PATCH v4]

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)

What is comming [PATCH v11]

Smaller MVP:

- remove file system support (i.e. inode map) for now
- add a memory protection hook

Memory protection hook

Handle memory-rights related syscalls

- mmap(2)
- munmap(2)
- mprotect(2)
- pkey_mprotect(2)

Rights

- PROT_READ
- PROT_WRITE
- PROT_EXEC
- PROT_SHARE

New BPF_LANDLOCK_MEM_PROT

Dedicated eBPF program context

```
1  struct landlock_ctx_mem_prot {
2    __u64 address;
3    __u64 length;
4    __u8 protections_current;
5    __u8 protections_requested;
6  };
```

What could come later (medium/long-term)

Future Landlock program types

fs_get

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

Future Landlock program types

fs_get

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

fs_ioctl

check 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...

Landlock: wrap-up

User-space hardening

- programmatic and embeddable access control
- designed for unprivileged[2] use
- apply tailored access controls per process
- make it evolve over time (map)

[2] If you can move mountains, you can move molehills.

Landlock: wrap-up

User-space hardening

- programmatic and embeddable access control
- designed for unprivileged[2] use
- apply tailored access controls per process
- make it evolve over time (map)

Current status

- standalone patches merged in net/bpf, security and kselftest trees
- current `security/landlock/*`: ∼1080 SLOC
- stackable security module
- ongoing patch series: LKML, `@l0kod`

[2] If you can move mountains, you can move molehills.

https://landlock.io

What about Kernel Runtime Security Instrumentation?

Goal

- framework to run security agents, i.e. HIDS (and HIPS?)
- mainly focused on malicious behavior detection

Common points

- LSM
- eBPF

Differences

- global system audit (neither by cgroups nor by process hierarchies)
- no access-control enforcement (for now)
- not designed for unprivileged use (for now)
- pretty new RFC/PoC