DEVCONF.cz

Seitan

A plant-based recipe against syscall anxiety

Alice Frosi
Principal Software Engineer

Stefano Brivio
Principal Software Engineer

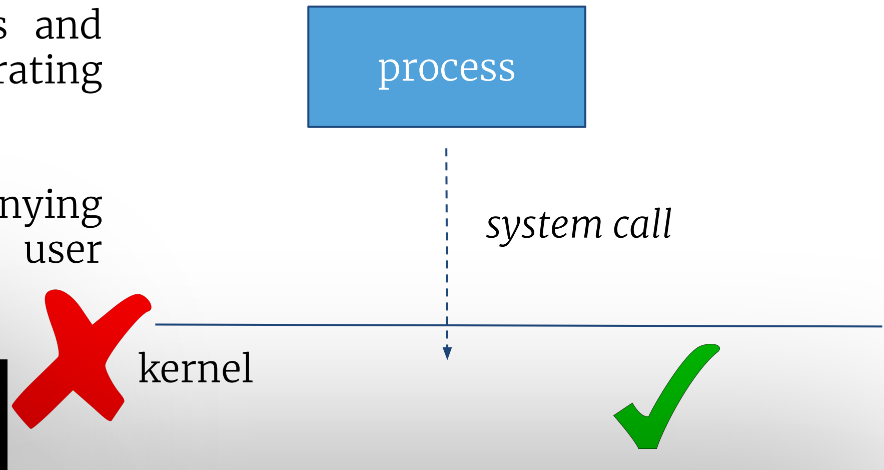
Agenda

1. Privileged operations, seccomp, containers
2. Seitan
3. Demo
4. Questions, and maybe answers!

System calls and privileged actions

System calls are the essential abstraction representing requests for system services and access to resources on most modern operating systems

A number of security models are based on denying or permitting system calls, depending on user privileges, capabilities, context, etc.




```
$ strace -e finit_module /sbin/modprobe evil_things
finit_module(3, "", 0)              = -1 EPERM (Operation not permitted)
modprobe: ERROR: could not insert 'evil_things': Operation not permitted
+++ exited with 1 +++
```

```
$ strace -e openat touch my_own_files
openat(AT_FDCWD, "my_own_files", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
+++ exited with 0 +++
```

What we want to improve

Enable users of container and virtualisation engines to grant fewer privileges to processes, with a mechanism to allow just the few privileged operations they need:

- Creating a tun device: `ioctl(…, TUNSETIFF, …)` requires `CAP_NET_ADMIN`, which implies complete control of network resources
- Setting scheduler policies for *one* process: `sched_setscheduler()` requires `CAP_SYS_NICE`, which can be used to CPU-starve *any* process

Enhance granularity and control over resource access mediated by system calls

- `CAP_MKNOD` is often granted to container engines, but it enables creation of any device node
- mount a specific volume: often via ad-hoc RPCs to avoid granting broad capabilities

Access control for resources:

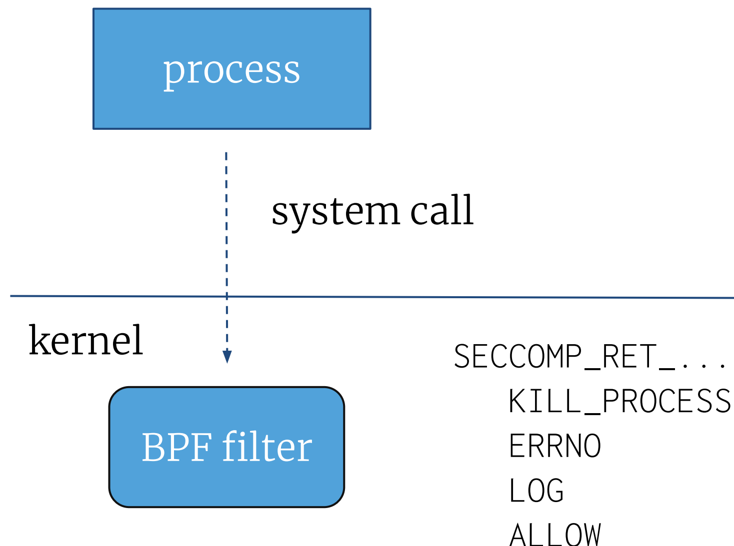
- connecting to privileged daemons, or opening files/devices, with per-container checks

State of the art

Seccomp BPF

Seccomp BPF (SECure COMPuting with Berkeley Packet Filters) is a Linux kernel feature offering basic system call filtering to reduce the exposed kernel surface available to applications

...but it can only accept, block, or log calls, and it doesn't dereference pointer arguments to process memory.



[Seccomp BPF (Secure Compute with filters)](#)

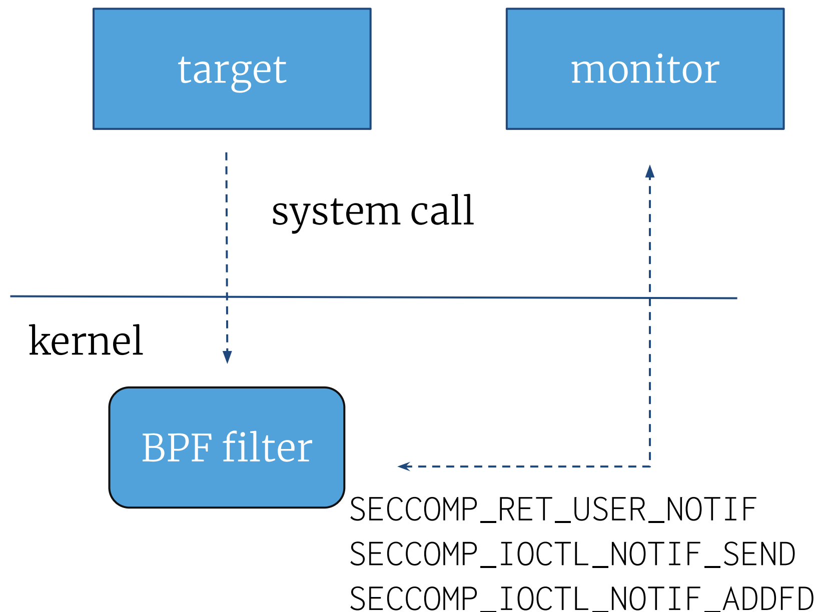
[seccomp(2)](#)

DEV CONF .CZ

Seccomp notifiers

Seccomp notifiers tell an userspace application about filtered system calls, along with their arguments.

The supervising process replies with return and error values, and tells the kernel if the system call should actually be issued. File descriptors can be added back into the calling process.



[The Seccomp Notifier - New Frontiers in Unprivileged Container Development, by Christian Brauner]()

[Seccomp user-space notification and signals [LWN.net]]()

[seccomp_unotify(2)]()

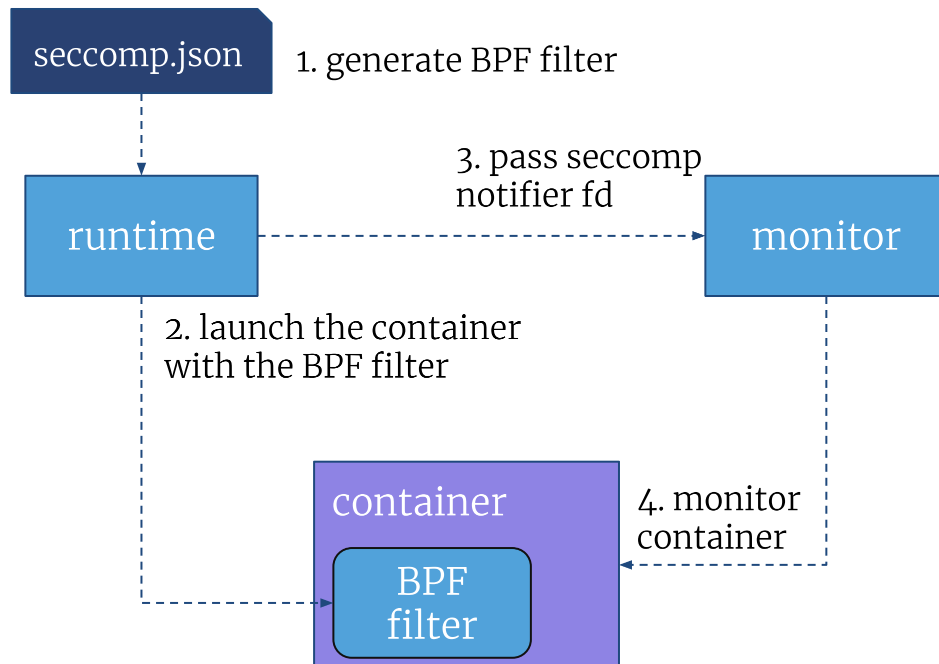
DEV CONF .CZ

Containers, OCI and k8s

Seccomp profiles are JSON files part of OCI spec defining the allowed, denied or notifiable syscalls

BPF filters are generated using libseccomp, based on the seccomp profile of the container

Support in OCI for seccomp notifiers with UNIX domain sockets



Existing solutions using seccomp notifiers

Existing solution using seccomp notifiers:

- LXD
- Kinvolk seccomp agent

What do they have in common:

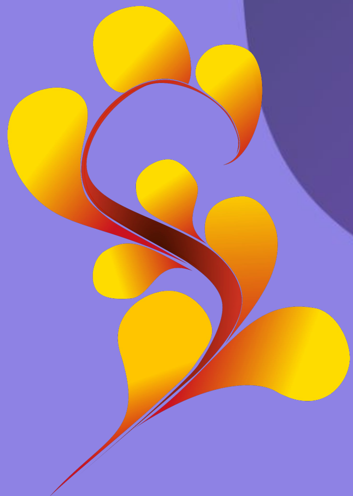
- Implement a seccomp notifier handler per syscall
- A new syscall or behavior ⇝ new code
- Not easily reusable
- Require understanding of seccomp notifiers

DEV CONF .CZ

Seitan

Syscall Expressive Interpreter, Transformer and Notifier

https://seitan.rocks/



Idea

- *recipes* describe matches of syscalls and arguments and corresponding action
- *seitan-cooker* follows the recipe and builds:
  - BPF program
  - gluten: a bytecode representation of matches and actions
- *seitan-eater* loads the filter and launches target process
- *seitan* loads the bytecode, monitors the notifier, matches on syscalls and executes actions

recipe

match: ioctl(TUNSETIFF) ↝ action1

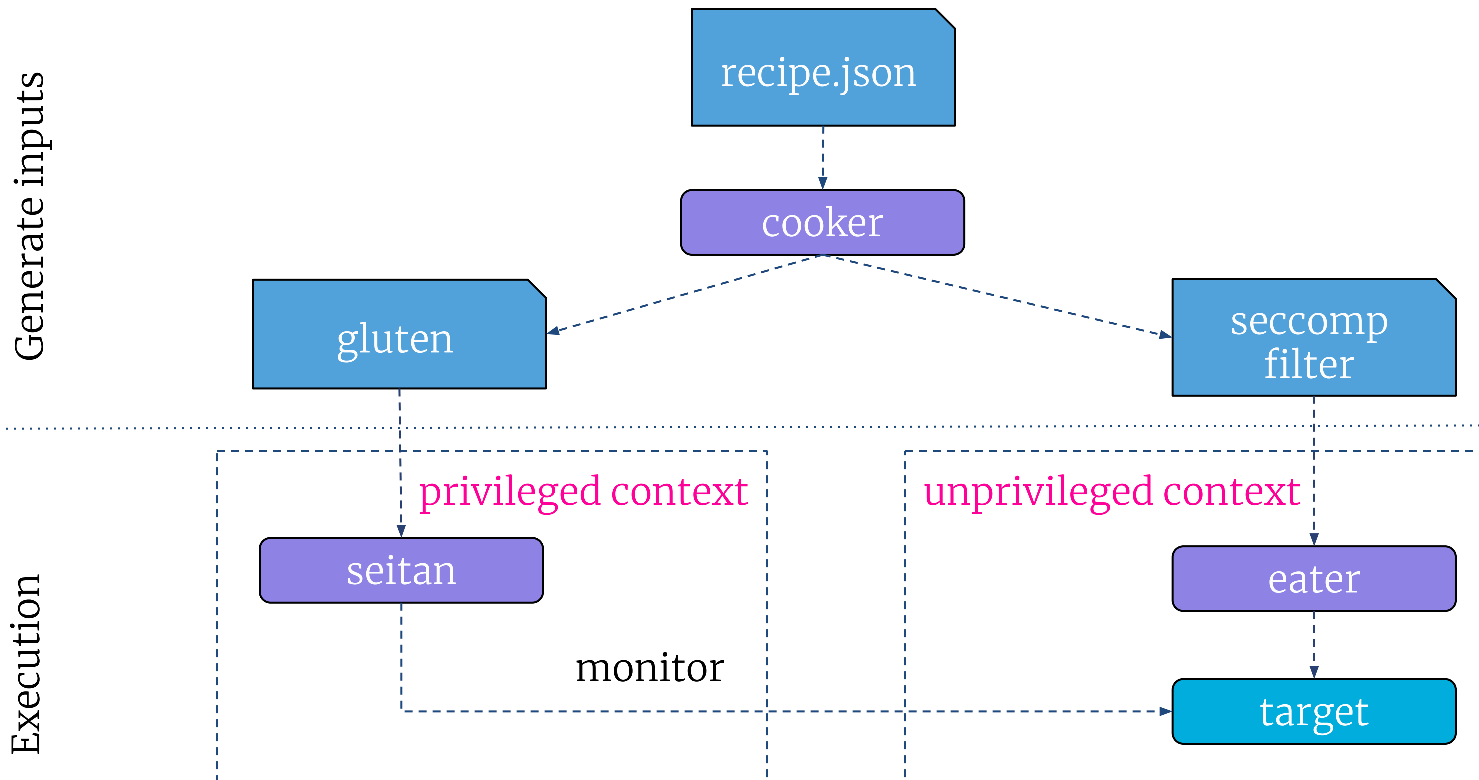
match: mknod(path) ↝ action2

match: mount(path) ↝ action3

match: connect(path) ↝ action4

DEV CONF .CZ

Flow



DEV CONF .CZ

Why seitan

- Declarative approach, not imperative
  - Improved visibility, single auditing point for privileged operations
- Flexible, with no extra coding necessary
  - Admins and tools only need to define the JSON recipe
- Generic
  - Independent and self-contained tool for specifications and generation of BPF programs and action bytecodes
  - *gluten* (bytecode) and BPF program can be generated and signed separately before running the workload

Why seitan

- Declarative approach, not imperative

```
for i, volume := range vmi.Spec.Volumes {
 if volume.ContainerDisk != nil {
		diskTargetDir, err := containerdisk.GetDiskTargetDirFromHostView(vmi)
		if err != nil {
			return nil, err
		}
		diskName := containerdisk.GetDiskTargetName(i)
		// If diskName is a symlink it will fail if the target exists.
		if err := safepath.TouchAtNoFollow(diskTargetDir, diskName, os.ModePerm); err != nil {
			if err != nil && !os.IsExist(err) {
				return nil, fmt.Errorf("failed to create mount point target: %v", err)
			}
		}
		targetFile, err := safepath.JoinNoFollow(diskTargetDir, diskName)
		if err != nil {
			return nil, err
		}
		sock, err := m.socketPathGetter(vmi, i)
		if err != nil {
			return nil, err
		}

		record.MountTargetEntries = append(record.MountTargetEntries, vmiMountTargetEntry{
			TargetFile: unsafepath.UnsafeAbsolute(targetFile.Raw()),
```

```
"match": {
  "openat": {
    "path": "/disk"
  }
},
"call": {
  "openat": {
    "path": "/mapped"
  },
  "ret": "fd"
},
"fd": {
  "src": { "tag": "fd" },
  "return": true
}
```

Use cases

Improving security posture by reducing privileges

- Rootless containers
  - Removing capabilities by impersonating only the necessary syscalls
- Argument introspection
  - Enable safe checks on dereferenced memory (strings, structs, buffers) through deep copy: arguments point to local copy, instead of original (race-prone) data
- Syscall counters
  - Fine grained control of process behaviour by counting syscall executions

Use cases

Testing

- Error injection on a syscall (e.g. return different error type)
- Mocking a particular syscall
- Inject a delay on a syscall (sleep + continue the syscall)

Application profiling

- Tracing syscalls executed by the target process

Resource allocation and management

- File descriptor injection, alternative way to `SCM_RIGHTS` and `pidfd_getfd(2)`
- Socket communication for containerised applications

DEV CONF .CZ

Example: impersonate a syscall

- Filtered syscall: mknod()
- Context: caller's mount namespace
- Action: replay mknod()
- Result: execute mknod only for a subset of minor numbers

```
"match": [
  { "mknod":
    {
      "path": { "tag": "path" },
      "mode": { "tag": "mode" },
      "type": { "tag": "type" },
      "major": 1,
      "minor": { "value": { "in": [ 3, 5, 7, 8, 9 ] }, "tag": "minor" }
    }
  }
],
"call":
  { "mknod":
    { "path": { "tag": { "get": "path" } },
      "mode": { "tag": { "get": "mode" } },
      "type": { "tag": { "get": "type" } },
      "major": 1,
      "minor": { "tag": { "get": "minor" } }
    },
    "context": { "mnt": "caller" }
  },
"return": { "value": 0, "error": 0 }
```

Example: syscall mocking and error injection

- Filtered syscall: connect()
- Result: pretend success on the first path, report permission denied on the second path

```
{
  "match": [
    { "connect": {
      "addr": {
        "family": "unix",
        "path": "/test1.sock"}
      }
    }
  ],
  { "return": { "value": 0, "errno": 0 } }
},
{
  "match": [
   { "connect": {
     "addr": {
       "family": "unix",
       "path": "/test2.sock"
     }
  ],
  { "return": { "value": 0, "errno": -1 } }
}
```

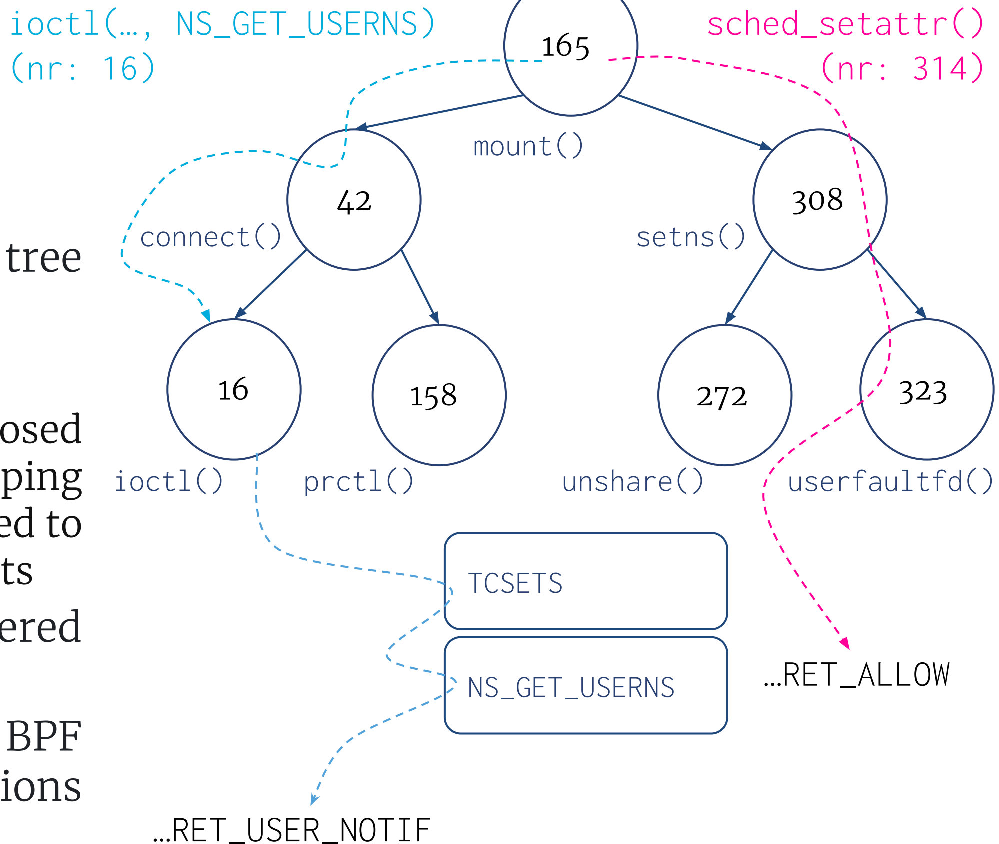
BPF program

The BPF program is a binary search tree indexed by system call number:

- search complexity, average: $\mathcal{O}(log\ n)$
- optimisation targets: many, as opposed to *libseccomp* simpler goal of keeping unfiltered calls fast. Notified calls need to be fast too: multiple terminal elements

Blocked syscall are treated as filtered syscalls: those can be slow.

Checking as much as possible in BPF program: numeric argument conditions sequentially linked to leaves



DEV CONF .CZ

Overhead

- Pushing most argument checks into BPF program: supervisor is used infrequently
  - no mandatory implementation of full syscall set (cf. gVisor, different goals)
  - we can do *a bit* better once *and if* eBPF becomes friends with seccomp
- We're rather on the control path, not so much on the data path
  - proxy as little as we can (access control), not I/O or packet transfers
- The filter means some overhead anyway. Do we care? Quick micro-benchmark on post-modern x86_64 laptop (don't quote us on this!)
  - baseline: 10M `lseek()` in 6.7s
  - BPF program attached, 100 unvisited instructions, match on `lseek()`, single compare and jump to the end, then `RET_ALLOW`: 10M `lseek()` in 8.2s
  - ~30ns per BPF instruction, 20-40 CPIs
  - ...I guess we don't care?

Revisiting eBPF Seccomp Filters, Jinghao Jia, Linux Plumbers Conference 2022

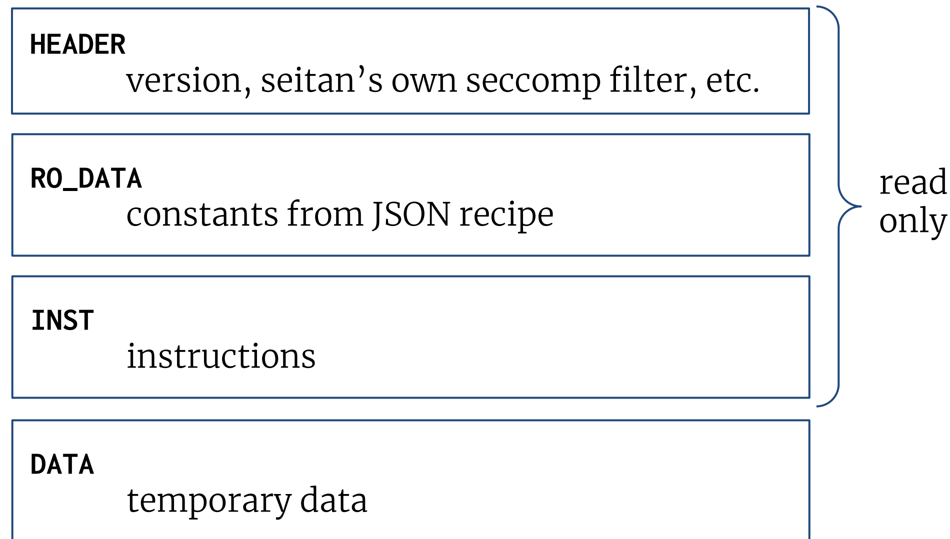
Bytecode memory layout

Seitan memory is statically allocated

HEADER, INST, and RO_DATA sections are filled with gluten bytecode as seitan starts

struct seccomp_data is set by the kernel on a seccomp notification

DATA section for copying data at runtime (struct, buffers, strings...)



Gluten and actions

OP_NR ⇝ jump to matches matching the system call number

OP_CALL ⇝ execute privileged syscall

OP_FD ⇝ inject a file descriptor (atomically)
OP_RETURN ⇝ set return and errno value or let the syscall go on

} seccomp notify replies

OP_COPY ⇝ copy an argument
OP_LOAD ⇝ load argument via `/proc/PID/mem`
OP_STORE ⇝ store data at pointer argument of process
OP_CMP ⇝ compare arguments
OP_BITWISE ⇝ logic operations

} memory operations

OP_RESOLVEFD ⇝ check if a file descriptor's inode matches a path

DEV CONF .CZ

System call context

The supervisor executes a system call on behalf of the target – with a fresh, verified copy of the arguments

Context specification:

- namespace (mount, network, PID, cgroup, etc.)
- working directory
- UID/GID

DEV CONF .CZ

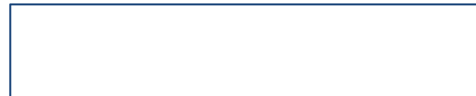
Tags

Set and get references between arguments (and conceptually distinct fields within arguments

Examples:

- a privileged system call creates a file descriptor used to replace the original descriptor in the target process
- derive arguments from original system call's arguments

x



```
{ “tag”: { "set": "x" } }

{ “tag”: { "get": "x" } }
```

DEV CONF .CZ

Security: how bad is it?

- Sometimes, all we need to do is to open a well-defined path from a different mount namespace – *not* to tell another component that it should open a given `../../../../path` from a different mount namespace and return a file descriptor corresponding to it
- Unified declarative approach to privileged operations: obvious benefit
- No parsing in supervisor, ~500 LoC, easy to audit, static memory only
- Surface: malicious JSON, malicious bytecode, malicious BPF program
- ...your concern here.

Demo

Takeaways

- Tool for filtering and executing privileged syscalls
- Capability and privileged reduction given to containers
- Declarative vs. imperative way
- Filtered syscalls with actions into a single file
- More (and more coming) at https://seitan.rocks

Future plans

- Finish modeling the system calls we want (maybe “all”, or maybe only 50-100 of them?), clean up code, man pages, packages…
- Get feedback on the idea right after this slide
- Offer seitan integration with container engines (e.g. Podman, cri-o, containerd…) and virtualisation engines (KubeVirt use cases)
- Extend Kubernetes to support already generated BPF filters

DEV CONF .CZ

Credits

- Andrea Arcangeli: originally wrote seccomp and told us this isn’t necessarily a bad idea, offered extensive feedback
- Christian Brauner: extended seccomp BPF with user notification and excellent documentation all along
- Ľuboslav Pivarc, Vladik Romanovsky (KubeVirt developers): feedback, endless discussions and encouragement

Q&A