



Scaling BPF LSM hooks and error injection across the kernel

Anton Protopopov, Cisco

LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026



Linux CVEs mitigation

- With AI's help CVE is a buzzword this year
- CVEs ::= “Common Vulnerabilities and Exposures”
- Technically it's just a set of bugs, marked up to gather stats and experiment
- In any case, vulnerabilities are there, so we want to be able to mitigate them
- [See Daniel's talk on shirudo](#)

How to mitigate Linux vulnerabilities

A few ideas:

- **Patch + Reboot** *[this is what we want to avoid in the first place]*
- **Disable individual modules** *[if built as a module and not yet loaded]*
- **Live patching: kpatch, kGraft, Canonical Livepatch, SUSE Live Patching, etc.** *[one can spot the problem here: vendor-lock]*
- **Configuration hardening, LSM SELinux/AppArmor, etc.** *[pre-AI era]*
- **BPF LSM** *[provides the biggest coverage across security_* hooks, etc., etc.]*

Standard example: Copy-Fail (CVE-2026-31431)

- As it is common today, a bug have a dedicated web-site <https://copy.fail/>
- In short, it allowed to gain root on any Linux system since forever (2017) by allowing an attacker to write arbitrary 4 bytes into cache of any file

Standard example: Copy-Fail (CVE-2026-31431)

- As it is common today, a bug have a dedicated web-site <https://copy.fail/>
- In short, it allowed to gain root on any Linux system since forever (2017)
- In BPF LSM the mitigation would looks something like this:

```
SEC("lsm/socket_create")
int BPF_PROG(copy_not_fail, int family, ... int kern, int ret)
{
    if (ret)
        return ret;

    if (!kern && family == AF_ALG)
        return -EPERM;

    return 0;
}
```

Standard example: Copy-Fail (CVE-2026-31431)

- As it is common today, a bug have a dedicated web-site <https://copy.fail/>
- In short, it allowed to gain root on any Linux system since forever (2017)
- In BPF LSM the mitigation would looks something like this:

```
SEC("lsm/socket_create")
int BPF_PROG(copy_not_fail, int family, ... int kern, int ret)
{
    if (ret)
        return ret;

    if (!kern && family == AF_ALG)
        return -EPERM;

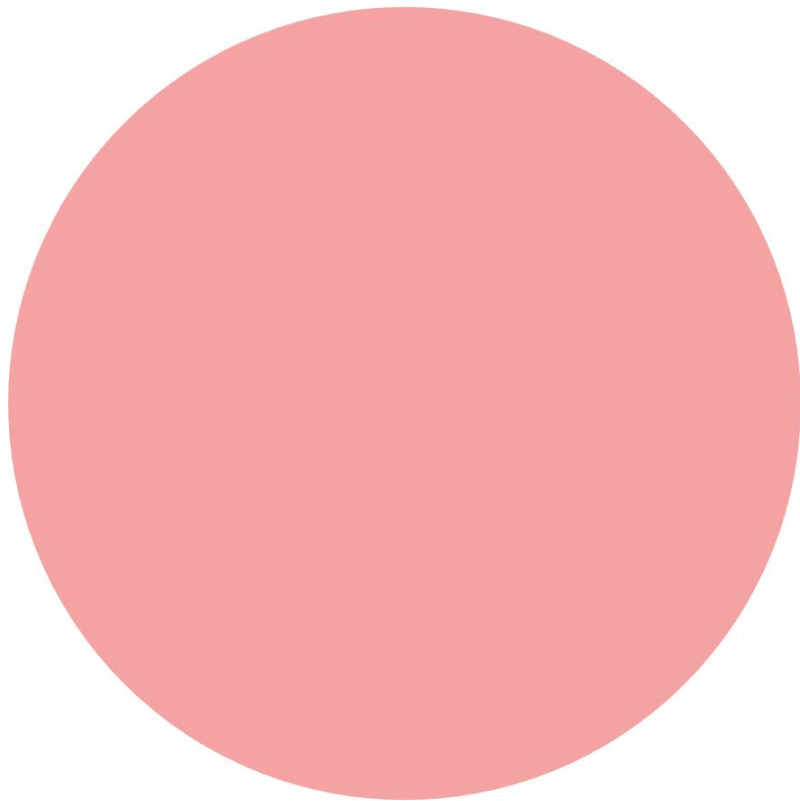
    return 0;
}
```

BPF-based protection (oss: tetragon, shirudo, etc.)

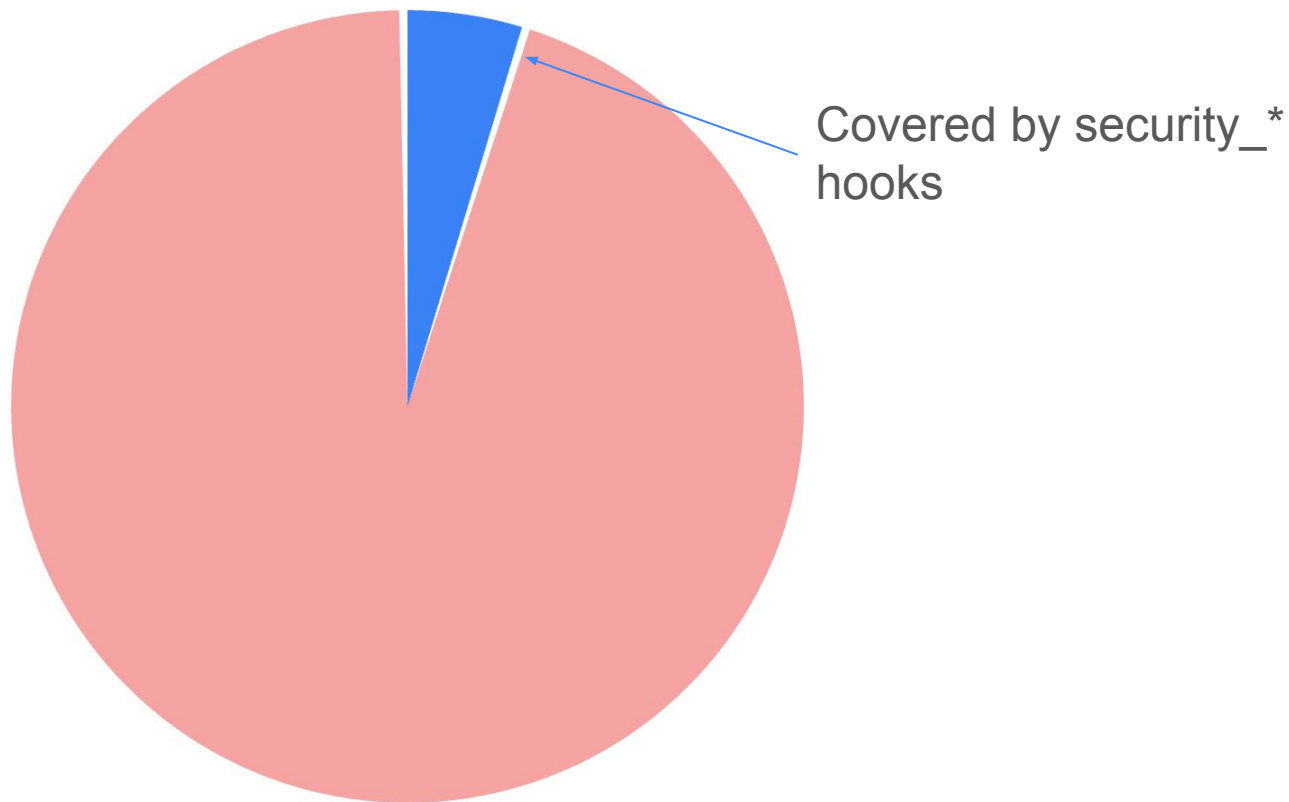
Overall idea:

- Agent-based or agentless setup
- Signed BPF loader + trust chain
- “Live-patching” new bugs with BPF
- (+ kill processes which try to do bad things)
- Looks good, sells good, but there is one small problem

Linux CVEs



Linux CVEs



Brilliant idea: let's add more hooks

The initial plan:

- **Analyse Linux CVEs** (the easiest source go gather stats)
- **Find common multiplexors** (most prominent candidates: nftables, net/sched, generic netlink, ethtool, ksmbd, ...)
- **Add new static [LSM, or LSM-like] hooks**

So, I've send [a series](#) for generic netlink and ethtool... *[see the next slide]*

Three subsystem maintainers commenting on my series



Back to the drawing board

- Besides the pushback from maintainers, another real problem is that finding new multiplexors helps to address additional 5-10% of CVEs (optimistically)
- What is a better way?

Killswitch

A Linux Killswitch was [proposed](#) this year by Sasha Levin:

- Let's just allow error injecting for any* function

Sounds like an ultimate solution, but honestly not realistic:

- No [standard] way to have a proper state, just echo “foo name” -> sysfs/...
- No way for a fine-grained control: say, we want to only reject some calls when flags != 0 or block only AF_ALG as for copy-fail

BPF: fmod_ret programs

We already have functionality which allows us to inject errors with BPF:

- For security_* hooks
- For syscalls
- For ALLOW_ERROR_INJECTION functions, if enabled

So, the proper BPF-way solution will be to [dramatically] extend the list of target functions and allow to error inject any **safe** function from this list

How does this look on practice?

Before getting into “how to get there”, let’s take a look at “what if we are there”

Best case, numbers, etc.

If we look at my test VM, and use new bpftool command to list fmodret progs:

```
# bpftool fmodret | head -4321 | tail -7; echo
1:123105    cxl_cper_sec_prot_err_valid (vmlinux)
1:123106    cxl_cper_setup_prot_err_work_data (vmlinux)
1:123121    cxl_reset_bus_function (vmlinux)
1:123143    d3cold_allowed_store (vmlinux)
1:123174    d_mark_tmpfile_name (vmlinux)
1:123186    d_set_mounted (vmlinux)
1:123210    da903x_probe (vmlinux)
```

```
# bpftool fmodret | wc -l
18073
```

How does this look on practice?

If we look at my test VM, and use new bpftool command to list fmodret progs:

```
# bpftool fmodret | head -4321 | tail -7; echo
1:123105    cxl_cper_sec_prot_err_valid (vmlinux)
1:123106    cxl_cper_setup_prot_err_work_data (vmlinux)
1:123121    cxl_reset_bus_function (vmlinux)
1:123143    d3cold_allowed_store (vmlinux)
1:123174    d_mark_tmpfile_name (vmlinux)
1:123186    d_set_mounted (vmlinux)
1:123210    da903x_probe (vmlinux)
```

```
# bpftool fmodret | wc -l
18073
```

Example

I've asked a robot to take a look at 10 random "high/critical" cves, and check if they can be mitigated with new attachment points

Example

I've asked one robot to select me 10 random "high/critical" cves, and then another robot to check if they can be mitigated with new hooks. **All ten are "yes"**

1.	CVE-2026-52943	YES	pskb_carve_inside_{header,nonlinear}
2.	CVE-2026-46319	YES	tcf_idr_check_alloc
3.	CVE-2026-23111	YES	nfnetlink_subsys_register
4.	CVE-2025-71089	YES	iommu_attach_device_pasid
5.	CVE-2025-21702	YES	tc_modify_qdisc
6.	CVE-2025-21692	YES	tc_ctl_tclass
7.	CVE-2024-50066	YES	__arm64_sys_mremap
8.	CVE-2024-43882	YES	notify_change
9.	CVE-2024-26898	YES	__register_chrdev
10.	CVE-2025-38585	YES	really_probe

Example: a small warning

But one shouldn't blindly believe robots, 2/10 mitigations are like this:

```
/* CVE-2026-23111: disable nftables registration, not all nfnetlink. */  
SEC("fmod_ret/nfnetlink_subsys_register")  
int BPF_PROG(block_nftables_registration,  
             const struct nfnetlink_subsystem *subsys, int ret)  
{  
    if (ret)  
        return ret;  
    if (BPF_CORE_READ(subsys, subsys_id) == NFNL_SUBSYS_NFTABLES)  
        return -EPERM;  
    return 0;  
}
```

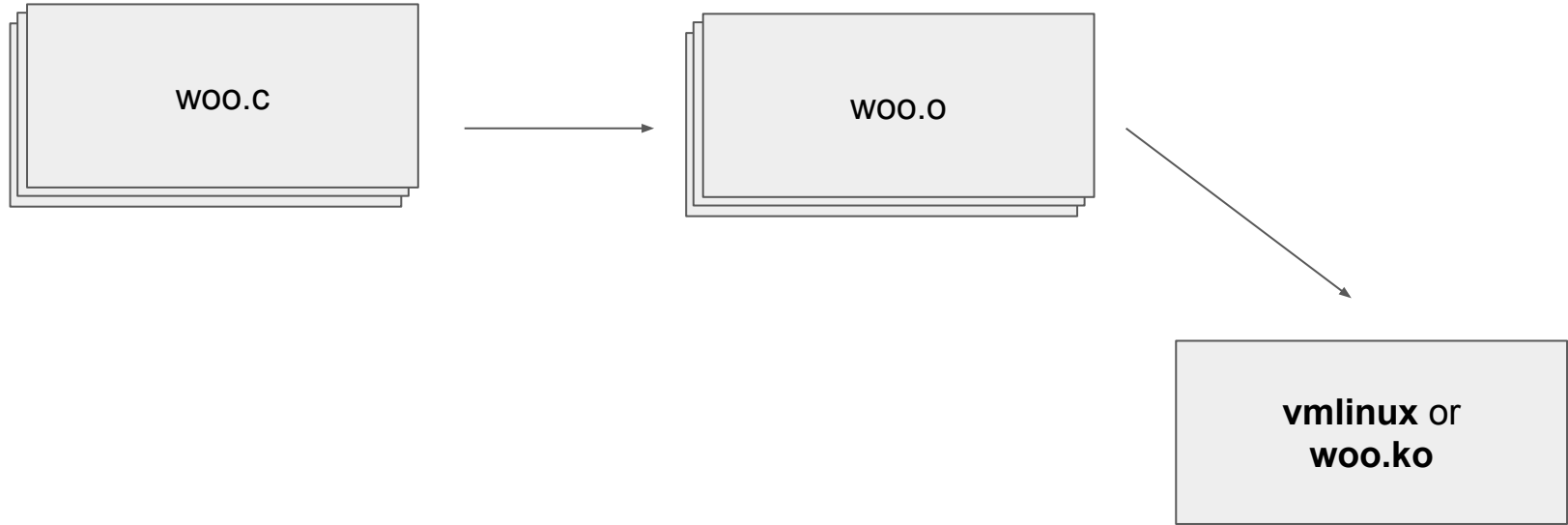
More examples I've checked

More examples I've validated:

- My previously proposed generic netlink “static hooks” are fully covered (all selftests pass)
- [Christian's fs/cgroups hooks proposed](#) (and not merged for similar reasons as mine):
 - Cgroup hook is super-easy, maps 1:1
 - Ns hooks, not 1:1, but individual ops->install (mntns_install, netns_install, ...) can be policed

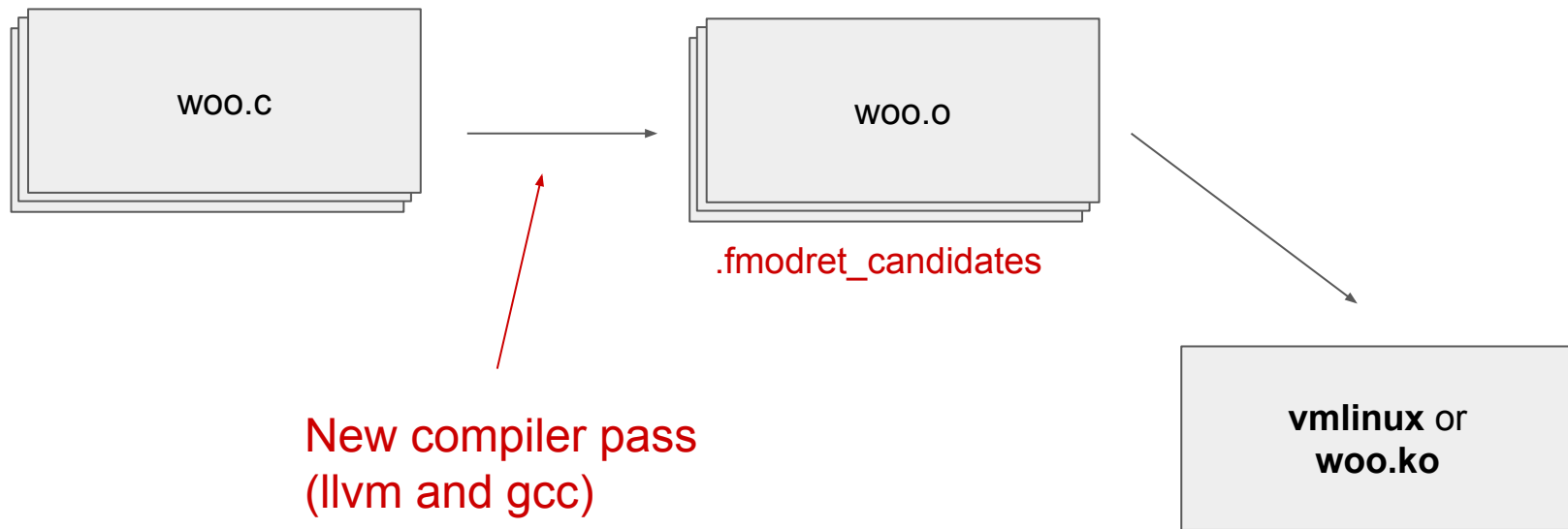
How to generate the list of safe functions?

Kernel/module build:



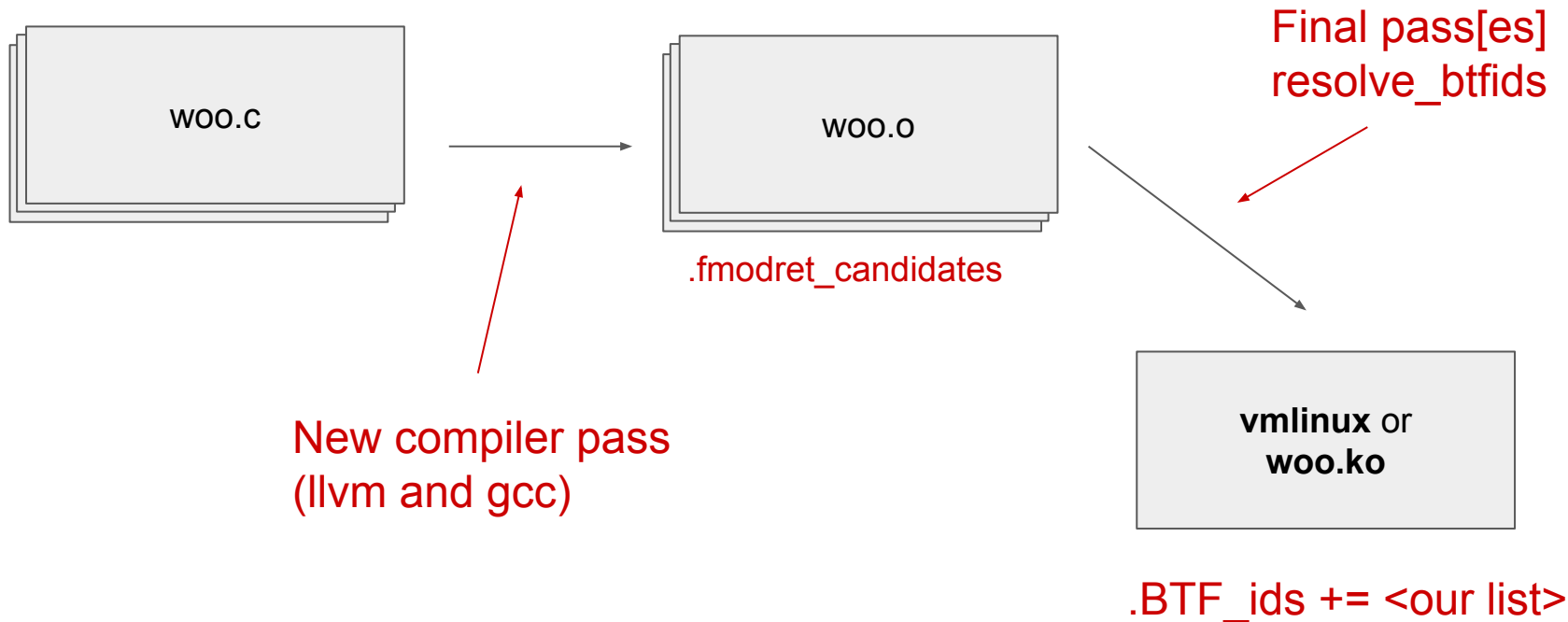
How to generate the list of safe functions?

Kernel/module build:



How to generate the list of safe functions?

Kernel/module build:



What is a “safe” function?

Let's stick to the existing description from [Documentation/fault-injection](#). Namely, a function is safe to be error-injected, if

- The function returns an error code if it fails
- Before returning error, the function does not generate any side-effects, such as taking and not releasing locks, io outside of stack, etc.
- The callers must check it correctly (need to recover from it)

Example: safe function

```
int __sys_socketsmth(int family, int type, int protocol)
{
    int flags;

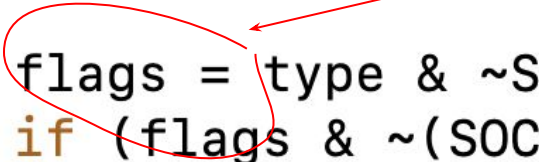
    flags = type & ~SOCK_TYPE_MASK;
    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
        return -EINVAL;

    ...
}
```

Example: safe function

```
int __sys_socketsmth(int family, int type, int protocol)
{
    int flags;
    flags = type & ~SOCK_TYPE_MASK;
    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
        return -EINVAL;
    ...
}
```

ok, write to stack



Example: another safe function

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
    int retval;
    struct file *file;

    file = file_close_fd(fd);
    if (!file)
        return -EBADF;
}
```

Example: another safe function

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
    int retval;
    struct file *file;

    file = file_close_fd(fd);
    if (!file)
        return -EBADF;
```

Takes a lock, but
always releases it



Example: unsafe function

```
int vfs_fchmod(struct file *file, umode_t mode)
{
    audit_file(file);
    return chmod_common(&file->f_path, mode);
}
```

side-effect



Safe functions

Here is a list of common safe function kinds:

- “Clear” path to a `return -ERRNO`
- “Clear” path to `return err`, where `err` \in `[-4095, -1]`
- “Clear” path to `if (IS_ERR(ptr)) return PTR_ERR(ptr)`
- “Clear” path to `err = moo(); if (err) return err`, where `moo()` is “safe”

Besides this, we can think of other types of functions not supported by BPF now:

- Functions, returning pointers with clear path to `return ERR_PTR(-ERRNO)`
- Functions, returning pointers with convention `NULL ⇒ error`
- ...

Why to check return value?

```
int f(void **obj) { /* alloc and assign *obj */ }

void g() {
    void *obj;
    f(&obj); /* can't fail in this path */
    deref(obj);
}
```



Boom

We also need to track checks like this

Our prog can't safely inject
EPERM

```
err = foo();  
if (err == -ENOMEM)  
    goto cleanup;  
do_smth_harmful();
```



Example: looking at real ALLOW_ERROR_INJECTION

```
/* If this is a leaf block and it is corrupt, just return -EIO. */  
if (found_level == 0 && btrfs_check_leaf(eb))  
    ret = -EIO;
```

Example: looking at real ALLOW_ERROR_INJECTION

```
if (btrfs_header_level(eb))
    ret = btrfs_check_node(eb);
else
    ret = btrfs_check_leaf(eb);

if (ret < 0)
    goto error;
```

Example: looking at real ALLOW_ERROR_INJECTION

```
bool should_fail_alloc_page(gfp_t gfp_mask, unsigned int order)
```

```
    if (!(*alloc_flags & ALLOC_NOLOCK) &&  
        should_fail_alloc_page(gfp_mask, order))  
        return false;
```

Remaining WIP: formally define “need to recover from it”

What is the real definition of this?

- Should we demand “**if (err < 0)**”, “**if (err)**”?
- Is error propagation enough?
- Should we track error propagation to user/kthread/...?
- (syscalls propagate to user, security_* may not)

Side-note: EXPORT_SYMBOL

Problem with EXPORT_SYMBOL:

- We can't see all the users during build
- Should EXPORT_SYMBOL(foo) imply that foo is error-injectable?
- There are examples in kernel where this is not true
- Should we just enforce **__must_check**?

Global error propagation

Say, we already know what is considered a safe caller, now we need to track calls.

One compiler pass is not enough, as we can only track error propagation of static functions. There are few options to track error propagation outside of an object:

- Non-LTO: generate lists of all static calls (foo -> bar1, foo -> bar2). Then we can build/follow function calls (for bar1 find all funcs which call it, validate)
- Thin-LTO. Looks similar
- Full LTO: ultimate power, we can do whatever

(This is also required to answer “is **foo** safe, if it has a clean path propagating return value of a global function **moo**”.)

Build overhead

On my test system build overhead is not too big so far

- Wall time: +1.5%
- CPU time: +0.2%

(This is a build in VM, without any optimizations.)

Next steps

Overall plan:

- Send kernel-side changes (minimal: just to pass/register new list properly)
- Add minimal implementation, proven to be correct
(this could be: only “clean static funcs” + “no address taken”)

Rinse, repeat:

- Generate more candidates, verify more sets of functions (global error propagation, __weak functions, lock/unlock tracking, “side-effectless” functions with clear paths (aka hooks), function returning ERR_PTR, ...)
- Further away: inline functions (600-700K instances $\Rightarrow$ thousands “hooks”)

Thanks! Questions?

