BPF: Let's see more LSMs

KP Singh

Implementing an in-tree LSM with BPF

SafeSetID LSM

- Gate UID transitions with a global allow list
- Simple, but can be simpler, more flexible
- So, how did it go?

Policy Input: Strings and Maps

`“<UID_from>:<UID_to>” or “<GID_from>:<GID_to>”`

- Implemented using a `BPF_MAP_TYPE_HASH_OF_MAPS`
- 2 maps for `UID` and `GID` policies
  - `UID` -> [set of allowed UIDs]
  - `GID` -> [set of allowed GIDs]

Any issues?

- A dynamically sized inner array would have been nice
- Static initialization of the array map
- Tried `BPF_F_NO_PREALLOC` and got an `EINVAL`
- Found this [patch]()
- Used `BPF_F_INNER_MAP`, works.
- However `bpftool dump map <id>` shows a bunch of zeroed entries
    - Maybe the iteration causes the allocation?

Implementing LSM Hooks

- Surprisingly easy

- Some wins:

    - `force_signal` could be easily replaced with `bpf_send_signal`
    - LSM hook logic could largely be kept the same
    - Custom logging FTW!

- And then, refcounting:

    - Needed to grab and drop a reference to `group_info`

```
__bpf_kfunc struct group_info` *bpf_group_info_acquire(struct
group_info *gi)
{
       return get_group_info(gi);
}

__bpf_kfunc void bpf_group_info_release(struct group_info *gi)
{
       put_group_info(gi);
}

BTF_ID_FLAGS(func, bpf_group_info_acquire, KF_ACQUIRE |
KF_RET_NULL)
BTF_ID_FLAGS(func, bpf_group_info_release, KF_RELEASE)
```

The verifier need to be told that `group_info` member of `cred` can be trusted

```
BTF_TYPE_SAFE_TRUSTED(struct cred) {
        struct group_info *group_info;
};
```

We'll need a lot more of these!

Loop bounds, are hard...

The sequence of 8193 jumps is too complex.

```
for (i = 0; i < ngroups; i++) {
    if (!id_permitted_for_cred(old,
                              (kid_t){
                                 .gid = new_group_info->gid[i]
                              }, GID))
}
```

```
bpf_loop(MAX_GROUPS, loop_cb, &loop_ctx, 0);


int loop_ctx(u32 i, struct loop_ctx *ctx) {

    [...]

    if (!id_permitted_for_cred(old,
                               (kid_t){
                                 .gid = new_group_info->gid[i]
                               }, GID))

}
```

R2 is ptr_group_info
invalid variable offset

Trick success rate 50%

```
if (ngroups > MAX_GROUPS)
    return -EPERM;

for (i = 0; i < ngroups; i++) {
    if (!id_permitted_for_cred(old,
                              (kid_t){
                                  .gid = new_group_info->gid[i]
                              }, GID))
}
```

Trick success rate 100% (so far)

```
for (i = 0; i < MAX_GROUPS; i++) {

    if (i > ngroups)
        break;

    if (!id_permitted_for_cred(old,
                               (kid_t){
                                   .gid = new_group_info->gid[i]
                               }, GID))
}
```

What is the community doing with BPF LSM?

```
LSM_HOOK(int, 0, userns_create, const struct cred *cred)
```

`+ BPF LSM =` A simple solution to a long standing problem.

Can't agree on what a container is?

No problem, flexible policy to the rescue!

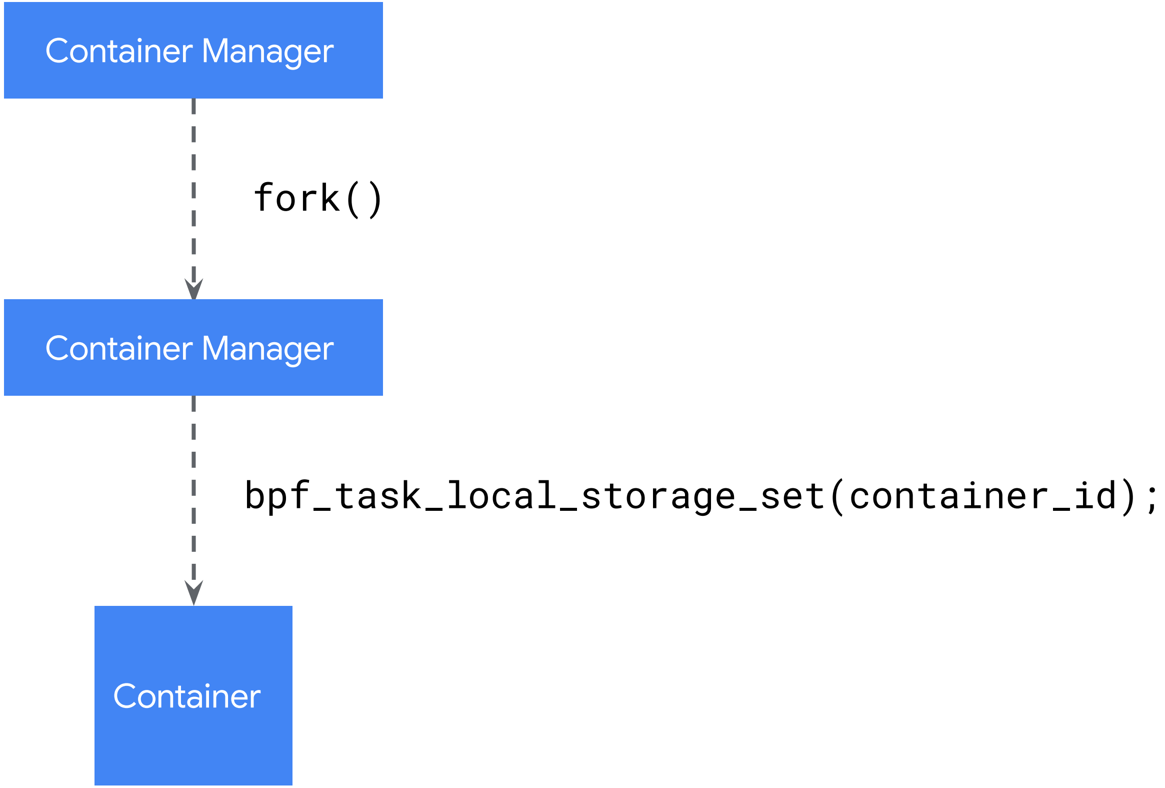
Container security

`lsm:task_alloc`

Propagate container ID forward

`lsm:bprm_check_security`

Custom security policy



Container Manager

`fork()`

Container Manager

`bpf_task_local_storage_set(container_id);`

Container

Systemd file-system restrictions

RestrictFileSystems=ext4 tmpfs

```
SEC("lsm/file_open")
int BPF_PROG(restrict_filesystems, struct file *file, int ret)
{

[...]
        int magic_number = file->f_inode->i_sb->s_magic);

        cgroup_id = bpf_get_current_cgroup_id();

        magic_map = bpf_map_lookup_elem(&cgroup_hash, &cgroup_id);
        if (!magic_map)
                return 0;

        if (bpf_map_lookup_elem(magic_map, &magic_number) == NULL)
                        return -EPERM;

        return 0;
}
```

Fix overhead: Almost there..

Summary

- LSM callbacks are indirect function calls
- Indirect function calls are susceptible to Branch target injection
- Retpolines are a security mitigation to prevent Branch Target Injection attacks
- Newer Intel CPUs added eIBRS, but with Branch History Injection being found last year. Retpolines are still needed.

Solution

We know the order and the list of LSMs at early boot

So, we don't really need indirect calls.

Just patch these call sites using static calls

The rest of the kernel is already doing it

[A lot of kernel code is patched by alternatives.c at early boot]

Okay, but what impact does it have?

Instructions	73,419,697	70,431,874
Branch Misses	**407,370**	**607,235**
Cache Misses	31,653	31,686
Branch Loads	**170,589,08**	**181,577,11**
Branch Load Misses	**407,388**	**607,253**

No really, what impact?

Benchmark	Delta (+ is better)
Execl Throughput	**+1.95%**
File Write 1024 bufsize 2000 maxblock	**+6.59%**
Pipe Throughput	**+9.55%**
Pipe-based Context Switching	**+3.02%**
Process Creation	**+2.33%**
Shell Scripts (1 concurrent)	**+1.49%**
System Call Overhead	**+2.78%**
System Benchmarks Index Score	**+3.49%**

Guess what is this?

600,000,000,000,000

Thank You!