kfuncs for BPF LSM use cases

LINUX PLUMBERS CONFERENCE | Vienna, Austria
Sept. 18-20, 2024

Who are we?

- My name is Matt Bobrowski
- I live in Zurich, Switzerland
- I work at Google on Security Endpoint Agents
- I have been dabbling in and around the Linux kernel for 5 or so years now
- I was recently appointed as the BPF LSM co-maintainer

- My name is Song Liu
- I live in Bay Area, California
- I work at Meta
- I am a maintainer/reviewer for various things in the kernel

What's new in the BPF LSM space?

We have some new sleepable BPF LSM hooks

- Added selected path-based (`CONFIG_SECURITY_PATH`) LSM security hooks to the pre-existing sleepable_lsm_hooks set, including:
  - `security_path_unlink()`
  - `security_path_mkdir()`
  - `security_path_rmdir()`
  - `security_path_truncate()`
  - `security_path_symlink()`
  - `security_path_link()`
  - `security_path_rename()`
  - `security_path_chmod()`
  - `security_path_chown()`
- Also added the new `security_file_post_open()` LSM security hook to the pre-existing sleepable_lsm_hooks set

And a bunch of other notable improvements here and there...

- We now have the BPF verifier performing improved return value range (`0, {1, -MAX_ERRNO}`) checking for BPF LSM programs
  - The BPF verifier now ensures that BPF LSM programs attached to a LSM hook returning:
    - `void` can return any value
    - A `bool-like` value can only return `0`, `1`
    - A possible error code value can only return `0, -MAX_ERRNO`
  - This improves overall system stability as we no longer need to concern ourselves with return values being misinterpreted from post LSM hook invocation
- Disallowed BPF LSM programs to attach to specific LSM hooks
  - Specifically, those which take output-like arguments as we can't exactly handle writes performed to those output arguments from the context of BPF LSM programs

New BPF kfuncs (currently) restricted to BPF LSM programs

- **bpf_get_file_xattr(struct file *file, const char *name__str, struct bpf_dynptr *value_p)**
  - KF_TRUSTED_ARGS, KF_SLEEPABLE
- **bpf_get_dentry_xattr(struct dentry *dentry, const char *name__str, struct bpf_dynptr *value_p)**
  - KF_TRUSTED_ARGS, KF_SLEEPABLE
- **bpf_get_task_exe_file(struct task_struct *task)**
  - KF_ACQUIRE, KF_TRUSTED_ARGS, KF_SLEEPABLE
- **bpf_put_file(struct file *file)**
  - KF_RELEASE
- **bpf_path_d_path(struct path *path, char *buf, size_t buf__sz)**
  - KF_TRUSTED_ARGS

Why the need for the new bpf_path_d_path() BPF kfunc?

- We're done wit the legacy `bpf_d_path()` BPF helper
  - It's inherently unsafe to use from a wide range of contexts
    - Naive usages of `bpf_d_path()` can lead to BPF programs being susceptible to memory corruption bugs
      - We've seen this issue with `bpf_d_path()` come up time and time again 1, 2, 3
  - Supplying an arbitrary pointer to a `struct path` buried away in some arbitrary in-kernel struct should simply not be permitted
  - We can now do better by enforcing more safety via `KF_TRUSTED_ARGS` semantics
  - Limited to sleepable BPF LSM(+) programs only, when technically, it can be called from non-sleepable contexts too
- Moving forward BPF LSM programs should use the `bpf_path_d_path()` instead, because it:
  - Enforces `KF_TRUSTED_ARGS` upon the struct path pointer supplied to it
  - Enforces that the supplied output buffer is sized correctly through the `__sz` argument annotations/constraints
  - No longer needs to be just used from sleepable BPF LSM programs

bpf_d_path()

```
SEC("lsm.s/file_open")
int BPF_PROG(file_open, struct file *file)
{
  int ret;
  char buf[64] = {};
  struct task_struct *current;

  current = bpf_get_current_task_btf();
  bpf_rcu_read_lock();
  ret =
bpf_d_path(&current->mm->exe_file->f_path,
           buf, sizeof(buf));
  /* Do something with buf */
  bpf_rcu_read_unlock();
  return 0;
}
```

bpf_path_d_path() (Preferred, use this!)

```
SEC("lsm/file_open")
int BPF_PROG(file_open, struct file *file)
{
  int ret;
  char buf[64] = {};
  struct file *exe_file;
  struct task_struct *current;

  current = bpf_get_current_task_btf();
  exe_file = bpf_get_task_exe_file(current);
  if (!exe_file)
    return 0;

  ret = bpf_path_d_path(&exe_file->f_path,
                        buf, sizeof(buf));
  bpf_put_file(exe_file);
  /* Do something with buf */
  return 0;
}
```

More on KF_TRUSTED_ARGS semantics

- kfuncs with with KF_TRUSTED_ARGS flag requires that all input pointers to BTF objects have been passed in their unmodified form
- Pointers passed directly to the BPF program as arguments are trusted (with some exceptions, check kernel/bpf/bpf_lsm.c:untrusted_lsm_hooks)
- A pointer returned by a KF_ACQUIRE BPF kfunc is considered as trusted by the BPF verifier
  - The verifier ensures that these pointers are released by a KF_RELEASE kfunc
- Pointer derived with pointer walking is not trusted (modulo some exceptions, check kernel/bpf/verifier.c:BTF_TYPE_SAFE_TRUSTED)
- Minimize the use of non-KF_TRUSTED_ARGS helpers/kfuncs

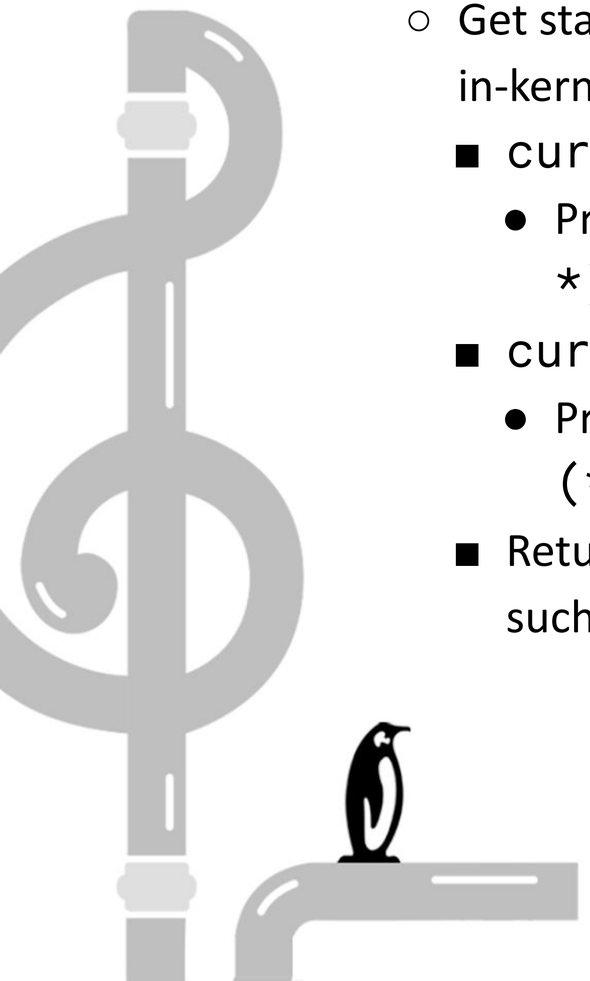
More on KF_TRUSTED_ARGS (cont'd)

```
SEC("lsm.s/file_open")
int BPF_PROG(hook_file_open, struct file *file)  /* file is trusted */
{
        struct task_struct *task = bpf_get_current_task_btf(); /* trusted */
        struct file *acquired;
        struct file *not_trusted;

        not_trusted = task->mm->exe_file; /* pointer walking, not trusted */
        acquired = bpf_get_task_exe_file(task); /* trusted */
        if (!acquired)
                return 0;

        bpf_put_file(acquired); /* acquired pointer must be released */
        return 0;
}
```

What's possibly coming up next for the BPF LSM?

We need more VFS-centric BPF kfuncs made available to the BPF LSM

- Some other things that we'd like to do from the BPF LSM are:
  - Get stable references to other nested struct path objects which buried away within some core in-kernel data structures, including:
    - `current->fs->root`
      - Proposal is to add something like `bpf_get_task_fs_root(struct task_struct *)`
    - `current->fs->pwd`
      - Proposal is to add something like `bpf_get_task_fs_pwd(struct task_struct (*)`
    - Returned `struct path` pointers can in turn be passed to things like `bpf_path_d_path()` such that reconstructed paths can be included within generated security events



We have some more specific use cases to discuss

Use case: Marking a set of files

- It is common for LSMs to specify a policy onto multiple files
- Requirement: handle large number of files
  - Files/subdirectory inherit property from parent directory
  - Some pattern/wildcard/regex can be really helpful
- Non-requirement: byte-to-byte verification
  - Checksum verifications
  - Signature checks
- `/usr/bin/*, /usr/bin/*/*, /dev/nvme[0-9]+n[0-9]+.*`

Solution 1: Label every file that matches the pattern with an xattr

- Pros
  - O(1) time overhead when checking the rules
- Cons
  - O(N) memory overhead, N=# of active inodes
- SELinux and Smack use this method
  - Set xattr in hook `inode_init_security()`
  - User space tools can also update xattrs
- Can we do this in BPF LSM? Not yet.
  - Hook `inode_init_security()` does not work for BPF LSM (more on this later)
  - setxattr is not allowed from BPF programs
  - xattr name prefix `security.bpf.*` is needed

Solution 2: Match full path to patterns

- Pros
  - Low memory overhead
- Cons
  - Expensive string operations needed to check path against rules
- Apparmor and Tomoyo use this method
- Can we do this in BPF LSM? Yes.
- Reconstruct path with `bpf_path_d_path()`
- No good pattern matching library in BPF (yet)

Solution 2: Match full path to patterns

- Pros
  - Low memory overhead
- Cons
  - Expensive string operations needed to check path against rules
- Apparmor and Tomoyo use this method
- Can we do this in BPF LSM? Yes.
- Reconstruct path with `bpf_path_d_path()`
- No good pattern matching library in BPF (yet)

Solution 3: Walk the VFS tree

- Pros
  - Low memory overhead
- Cons
  - No protection against race conditions (with rename, etc.)
- Landlock uses this approach
- Can we do this in BPF LSM? Yes, but we cannot (yet) use KF_TRUSTED_ARGS.

Solution 3: Walk the VFS tree (cont'd)

```
SEC("lsm/file_open")
int BPF_PROG(hook_file_open, struct file* file) {
    struct mount* mnt = container_of(file->f_path.mnt, struct mount, mnt);
    struct dentry* dentry = file->f_path.dentry;
    for (i = 0; i < MAX_WALK_DEPTH; i++) {
        struct dentry* root_dentry = BPF_CORE_READ(mnt, mnt.mnt_root);
        struct dentry* parent;
        if (ctx->dentry == root_dentry) {
            /* mount handling omitted for simplicity, something like follow_up() */
        }
        bpf_strncmp(rule_str[i], dentry->d_name, ...);
        parent = BPF_CORE_READ(dentry, d_parent);
        if (parent == dentry)
            break;
        dentry = parent;
    }
}      /* Note: mnt, dentry, parent are not trusted. */
```

Solution 4: Mark inode as we walk down VFS tree [1]

- How to do this in BPF LSM?
  - Load inode flags from xattr to BPF map on `security_d_instantiate()`
    - This is because we need dentry to read xattr (in fact, only special fs like 9p uses dentry for xattr)
  - Propagate inode flags to children on `security_inode_init_security()`
  - Check inode flags on `security_file_open()`
- Pros:
  - O(1) time overhead when checking the rules
- Cons:
  - O(N) memory overhead
- What is missing?
  - d_walk() like kfunc to update BPF map when xattr changes

[1] Based on https://lore.kernel.org/bpf/20240729-zollfrei-verteidigen-cf359eb36601@brauner/

Walk the VFS tree with trusted pointers

- Add more KF_ACQUIRE/KF_RELEASE based BPF kfuncs
- We need BPF kfuncs that operate on `struct dentry` i.e.
  - `bpf_dget()`, `bpf_dput()`, `bpf_dget_parent()`, `bpf_d_find_alias()`
- We need BPF kfuncs that operate on `struct mount` and `struct vfsmount`, i.e.
  - `bpf_mntget()`, `bpf_mntput()`, `bpf_real_mount()`
- Perhaps also RCU flavor kfuncs (KF_RCU_PROTECTED)
- And perhaps more...

- These new BPF kfuncs will be used to enforce trusted pointer semantics
- The BPF verifier will ensure that reference acquired by these kfuncs will be released

Walk the VFS tree with trusted pointers (cont'd)

```
SEC("lsm/file_open")
int BPF_PROG(hook_file_open, struct file* file) {
    struct mount* mnt = bpf_real_mount(file->f_path.mnt);
    struct dentry* dentry = bpf_file_dentry(file->f_path.dentry);
    for (i = 0; i < MAX_WALK_DEPTH; i++) {
        struct dentry* root_dentry = BPF_CORE_READ(mnt, mnt.mnt_root);
        struct dentry* parent;
        if (ctx->dentry == root_dentry) {
            /* mount handling omitted for simplicity */
        }
        bpf_get_dentry_xattr(dentry, "security.bpf.xxx", ...);
        parent = bpf_dget_parent(dentry);
        if (parent == dentry)
            break;
        bpf_dput(dentry);
        dentry = parent;
    }
    bpf_dput(dentry);
    bpf_mntput(mnt);
    /* Note: mnt, dentry, parent are trusted. */
```

Walk the VFS tree with trusted pointers and BPF iterator

- BPF iterators enable safe traversal of kernel objects
- We have some pre-existing BPF iterators already:
  - `task`, `task_file`, `task_vma`
  - `socket`
  - `map`, `map element`
  - `ksym`
- BPF iterator to walk dentry toward root
- BPF iterator similar to `d_walk()`

Walk dentry tree with trusted pointers and BPF iterator (cont'd)

```
SEC("lsm/file_open")
int BPF_PROG(hook_file_open, struct file* file) {
    bpf_for_each(dentry, dentry, &file->f_path, BPF_DENTRY_ITER_TO_ROOT) {
         bpf_get_dentry_xattr(dentry, "security.bpf.xxx", &value_ptr);
         /* check xattr in value_ptr */
    }
    ...
}
```

Align the BPF LSM more closely with other in-kernel LSMs

- BPF LSM is not yet as capable as in-kernel LSMs

- Missing per object data for some data types
- Not able to write to output arguments of LSM hooks

BPF local storage for per object data

- Most LSMs use a blob allocated for per object data
- BPF uses local storage, task local storage, inode local storage, etc.
- Still missing local storage for the following types
  - `struct file`
  - `struct cred`
  - `struct ipc`
  - `struct msg_msg`
  - `struct superblock`

LSM hooks with output arguments

- Some security hooks use pointer arguments for output
  - `security_inode_init_security()`
  - `security_sb_set_mnt_opts()`
  - `security_cred_getsecid()`
  - `security_current_getsecid_subj()`
  - `security_task_getsecid_obj()`
  - `security_ipc_getsecid()`
  - `security_getselfattr()`
  - `security_getprocattr()`
  - `security_secctx_to_secid()`
- Unlike in-kernel LSMs, the BPF LSM currently cannot write to these output pointer arguments
- Potential solutions
  - Add kfuncs for specific use cases
  - Create writable contexts for these output pointers

Thanks for your attention!
Questions?