Systems & Software Security

COMSM0050

2020/2021

University of BRISTOL

Linux Security Module Framework

How do we bring Linux to grade B?

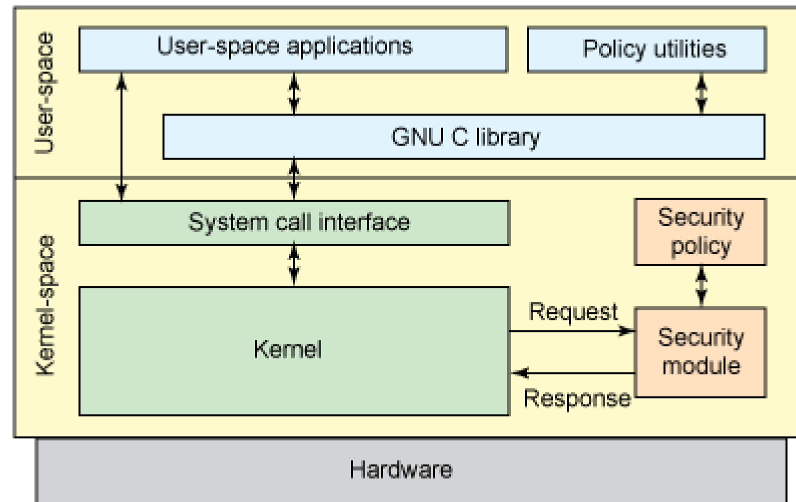
- This was/is a real research problem
  - See reading material on the course website
- Need to provide mechanism to implement one’s MAC policy
- Need to be small and testable

How do we bring Linux to grade B?

- This was/is a real research problem
  - See reading material on the course website
- Need to provide mechanism to implement one's MAC policy
- Need to be small and testable

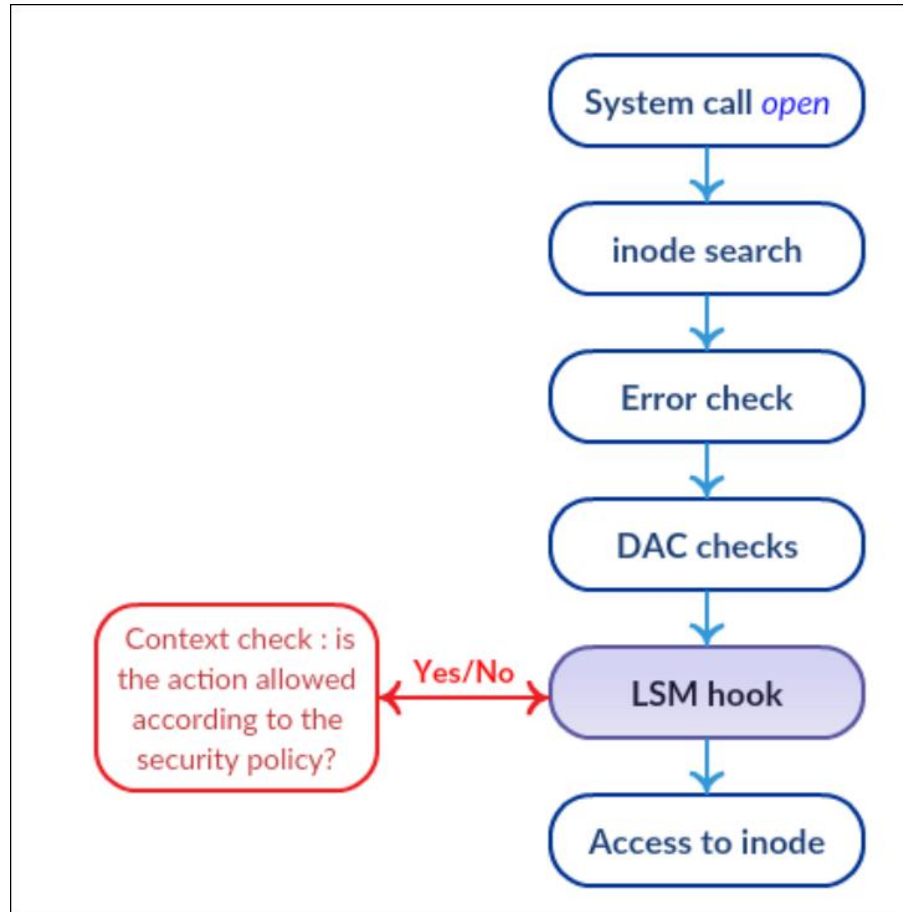
SOLUTION: Linux Security Module Framework

LSM Framework



- LSM implement the reference monitor concept
- Hooks call functions provided in a security module on interactions between subjects and objects
- Framework is verified
- Modules themselves should be small and verifiable

LSM Framework



- Security Hooks are invoked on the path between any subject action and an object
- Hook functions return access decision
  - 0 -> access
  - ENOMEM -> no memory
  - EACCESS -> access denied
  - EPERM -> privilege are required
- Hook function should be small
  - To be verifiable (security)
  - On critical path (performance)

SELinux hook implementation example

```
3527  static int selinux_file_permission(struct file *file, int mask)
3528  {
3529          struct inode *inode = file_inode(file);
3530          struct file_security_struct *fsec = selinux_file(file);
3531          struct inode_security_struct *isec;
3532          u32 sid = current_sid();
3533
3534          if (!mask)
3535                  /* No permission to check.  Existence test. */
3536                  return 0;
3537
3538          isec = inode_security(inode);
3539          if (sid == fsec->sid && fsec->isid == isec->sid &&
3540              fsec->pseqno == avc_policy_seqno(&selinux_state))
3541                  /* No change since file_open check. */
3542                  return 0;
3543
3544          return selinux_revalidate_file_permission(file, mask);
3545  }
```

```
3514  static int selinux_revalidate_file_permission(struct file *file, int mask)
3515  {
3516          const struct cred *cred = current_cred();
3517          struct inode *inode = file_inode(file);
3518
3519          /* file_mask_to_av won't add FILE__WRITE if MAY_APPEND is set */
3520          if ((file->f_flags & O_APPEND) && (mask & MAY_WRITE))
3521                  mask |= MAY_APPEND;
3522
3523          return file_has_perm(cred, file,
3524                               file_mask_to_av(inode->i_mode, mask));
3525  }
```

Type of hooks

- Managements hooks
  - Called to handle object life cycle
  - e.g. *security_inode_allocate* or *security_inode_free*
  - Used to manage security information
  - … in particular security blob (data that can be associated to subjects and objects)
- Path based hooks
  - Related to pathnames
  - Used to implement named based policies (B1 from previous video)
- Object based hooks
  - Path kernel structure corresponding to objects
  - Used to implement B2 type policies

Security pseudo file system

- Need to interact with the security module from userspace
  - Loading or editing access rules
  - Reading some audit data
  - Modify module configuration
- Achieved through a standard file interface

SELinux implementation example

```c
120 #define TMPBUFLEN       12
121 static ssize_t sel_read_enforce(struct file *filp, char __user *buf,
122                                 size_t count, loff_t *ppos)
123 {
124         struct selinux_fs_info *fsi = file_inode(filp)->i_sb->s_fs_info;
125         char tmpbuf[TMPBUFLEN];
126         ssize_t length;
127 
128         length = scnprintf(tmpbuf, TMPBUFLEN, "%d",
129                            enforcing_enabled(fsi->state));
130         return simple_read_from_buffer(buf, count, ppos, tmpbuf, length);
131 }
```

SELinux

- Implement MAC
- Combine two approaches popular in the 90s:
  - Type Enforcement (TE)
  - Role Based Access Control (RBAC)
- Labels all subjects with a security context
  - User
  - Domain
  - Role
- Rules describe what each subject domains can do to an object domains

SELinux example

- /etc/passwd contains information that should be read by any user
- /etc/shadow contains sensitive information

SELinux example

allow user_t public_t : file read

- user_t: “normal user”
- public_t: “normal object”

Normal users are allowed to read normal files

SELinux example

```
allow passwd_t passwd_data_t : file {read write}
```

Users in passwd_t domain can have read/write access to file in the passwd_data_t domain.

SELinux example

allow user_t passwd_exec_t : file execute

allow user_t passwd_t : process transition

Allow normal users to actually execute the passwd program

Second rules allows the domain to transition when passwd_t is executed.

SELinux example

type_transition user_t passwd_exec_t : process passwd_t

The rules that actually makes the transition happen.

SELinux example

- /etc/passwd contains information that should be read by any user
- /etc/shadow contains sensitive information

allow user_t public_t : file read

allow passwd_t passwd_data_t : file {read write}

allow user_t passwd_exec_t : file execute

allow user_t passwd_t : process transition

type_transition user_t passwd_exec_t : process passwd_t

University of BRISTOL

What do you think?

What do you think?

- This is extremely complicated!
- Available in Linux distributions (you can try it out on Fedora)
- NSA provides a reference policy
- It is hard to live with
    - Policy of around 20,000 rules
- Most popular SELinux query?
    - How to turn off SELinux
- Mostly appropriate for systems running fixed set of applications
    - Server
    - … used in Android too
- Rule design is a non-trivial task