Inside the Linux security module (LSM)

Vandana Salve, Prasme Systems

- Software architect and co-founder of Prasme systems, and Linux Trainer@Linux Foundations.
- Focus: Developement of system software for embedded devices and Linux systems

Vandana has involved in Linux system product development across number of domains including Embedded and Network server systems. She enjoys developing and teaching Linux systems, device drivers using the latest methods and tools.

1 Introduction to LSM

2 Digging deeper into the architecture

3 Integration of an LSM into the linux kernel

4 Types of LSMs

Why Linux security module

- Security is a chronic and growing problem: as more and more systems go on line, the motivation to attack rises and Linux is not immune to this threat:
  - Linux systems do experience a large number of software vulnerabilities.
- An important way to mitigate software vulnerabilities is through effective use of access controls
- The Linux Security Modules (LSM) seeks to solve this problem by providing a general purpose framework for security policy modules.

You will learn:

- To understand the LSM framework, its architecture and the existing LSM implementations

Introduction to Linux security module

- Lets understand what is Linux security module
  - Linux security module (LSM) is the framework integrated into the kernel to provide the necessary components to implement the Mandatory access control (MAC) modules, without having the need to change the kernel source code every time
  - Application whitelisting has been proven to be one of the most effective ways to mitigate cyber-intrusion attacks.
  - A convenient way to implement this widely recommended practice is through the ***"Linux Security Modules"***

What is LSM?

- An LSM is a code compiled directly into the kernel that uses the LSM framework.

- The LSM framework is intended to allow security modules to lock down a system by inserting checks whenever the kernel is about to do something interesting.

- The security modules hooks into those check points and for each operation, checks whether the operation is allowed by the security policy currently enforced or not.

- The LSM framework can deny access to important kernel objects,

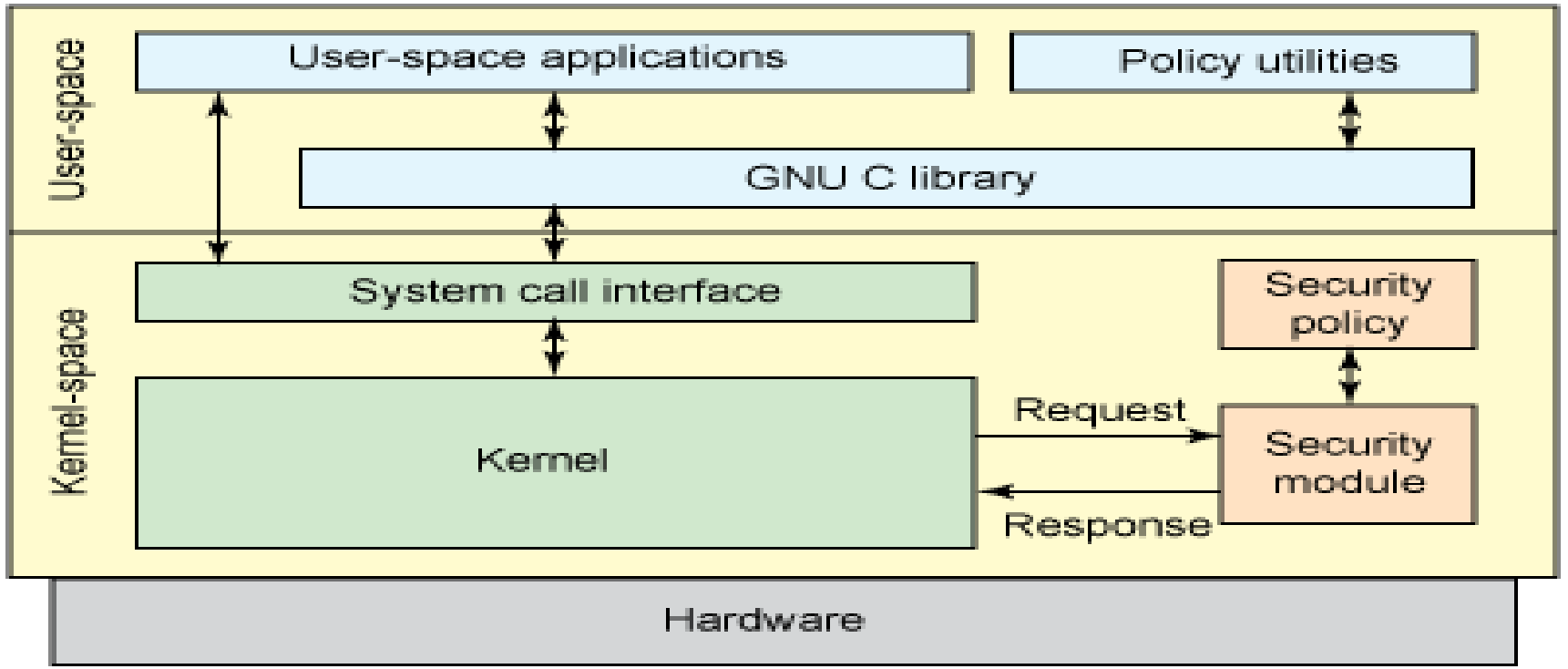
such files, inodes, task structures, credentials, and inter-process communication objects.

Major and Minor LSMs

- Major
  - The major LSMs are all implementations of MAC with configurable policies loaded from the user space
  - Only single LSM can be used at a time as they all assumed they had exclusive access to the security context pointers and security identifiers embedded in protected kernel objects
  - Examples: SELinux, SMACK, AppArmor and TOMOYO
- Minor
  - Minor LSMs implements a particular security functionality and are stacked on top of Major LSM and mosty need less security less context.
  - The minor LSMs typically only contain flags to enable/disable options as opposed to having policy files that are loaded from the user space as part of the system boor up.
  - Examples: Yama, loadPin, SetSafeID and Lockdown

Overview of LSM Framework

- The LSM framework provides a modular architecture that provides “hooks” built into the kernel and allows security modules to be installed, reinforcing access control.



Architecture of LSM

- The LSM framework allows third-party access control mechanisms to be linked into the kernel and to modify the default DAC implementation.
- By default the framework does not provide security in itself, it provides infrastructure to support the security modules.
- The LSM framework provides
  - Additional security fields(void *) to the kernel data structures.
  - Functionality to insert calls to the hook functions at critical points in the kernel code to manage the security fields and to perform access controls.
  - It also provides functions to register and un-register security modules

LSM Hooks

- Security hooks helps to mediate various operations in the kernel
  - These hooks invoke functions defined by the chosen modules
  - These hooks construct “authorization queries” that are passed to the module
  - The function calls that can be overridden by security module to manage security fields and mediate access to kernel objects.

LSM Security data fields

- LSM framework enables security modules to associate security information to kernel objects
- LSM extends “sensitive data types” with opaque data security fields
- The LSM security fields are simply void * pointers added in various kernel data structures
- They are completely managed by security modules

Security data fields inside kernel objects

- For process and program, security field added to
  - struct task_struct and struct linux_binrpm
- For File system, security field added to
  - struct super_block
- For pipe, file and socket, security field added to
  - struct inode and struct file
- For packet and network device, security field added to
  - struct sk_buff and struct net_device
- For System V IPC, security field added to
  - struct kern_ipc_perm and struct msg_msg

Examples of security data fields

```
struct file {
/*...*/

#ifdef CONFIG_SECURITY
        void                                    *f_security;
#endif
};

struct inode {
/*...*/

#ifdef CONFIG_SECURITY
        void                                    *i_security;
#endif
};

struct super_block {
/*...*/

#ifdef CONFIG_SECURITY
        void                                    *s_security;
#endif
};
```

LSM security data structures and hooks

- LSM data structure” struct security_hook_list”
  - This data structure maintains list of pointer to the security_hook_list and store information of LSMs added into the system.
- LSM data structure “union security_list_options”
  - Union of function pointers of the security hooks defined for the LSM, which are called at various critical paths in the kernel code.
- LSM data structure “structure security_hook_heads ”
  - This data structure containing the heads of the linked list corresponding to each hook, thus allowing them for execution in the right order, respecting the stacking property of LSM.

Looking in secutiy_list_options, hooks

- In the code snippet, we can see the hooks related to the creation and removal of directories, file open, inode/socket creation,task alloc, IPC etc
- int (*path_mkdir)(const struct path *dir, struct dentry *dentry,
  umode_t mode);
- int (*path_rmdir)(const struct path *dir, struct dentry *dentry);
- int (*file_open)(struct file *file);
- int (*inode_create)(struct inode *dir, struct dentry *dentry,
  umode_t mode);
- int (*socket_create)(int family, int type, int protocol, int kern);
- int (*task_alloc)(struct task_struct *task, unsigned long clone_flags);
- int (*ipc_permission)(struct kern_ipc_perm *ipcp, short flag);

Clarification about the LSM hooks

- Most of the hooks provided by LSM need to return an integer value (some return void)
  - “0” equivalent to the authorization
  - ENOMEM , No memory available
  - EACCESS, Access denied by the security policy
  - EPERM, Privileges are required to do this action
- Hooks provided by LSM can be 2 different types

  Object based hooks : these are related to kernel objects such C structures like inodes, files or sockets
  - Access authorization will be based on these object attributes
- Path based : these are related to paths

Clarification about security data fields/blobs

- It is the functionality provided by the LSM framework which allows for enabling special fields located in various kernel structures and reserved for use of security modules
- Their names usually ends with “_security” suffix.
- This allows for maintaining a context between different hooks, clearing the way for higher-level security policies

Only major LSMs can benefit of security blobs, Minor LSMs do not use security blobs

Other LSM features

- Aside from these hooks and the actions they permit, LSM framework also provide “Audit” functionality
- They provide alternative ways of generating log files
- LSM framework also supports the creation of pseudo-filesystems
  - to easily interact with the security modules from the user space
  - Loading and editing some access rules, reading some audit data or modifying the module's configurations

Integration of an LSM into the linux kernel

- Enabling the LSM in the kernel requires the following feature support
  - Kernel configurations
  - Makefiles
  - Changes to basic code security module
  - Integration with the LSM framework

LSM kernel configurations

- CONFIG_DEFAULT_SECURITY configuration needed to be selected at built time
- For the supported LSM, the configuration options are defined in "security/LSM-name/Kconfig"

```
config SECURITY_YAMA
        bool "Yama support"
        depends on SECURITY
        default n
        help
          This selects Yama, which extends DAC support with additional
          system-wide security settings beyond regular Linux discretionary
          access controls. Currently available is ptrace scope restriction.
          Like capabilities, this security module stacks with other LSMs.
          Further information can be found in
          Documentation/admin-guide/LSM/Yama.rst.

          If you are unsure how to answer this question, answer N.
```

LSM Makefile

- Makefile is also required to get the code compiled.
- For a simple minor LSM like Yama, it can look like that:

- obj-$(CONFIG_SECURITY_YAMA) := yama.o
- yama-y := yama_lsm.o
- Just as Kconfig files, Makefiles are organized as a tree and the following lines need to be added in the security/Makefile file:
- subdir-$(CONFIG_SECURITY_YAMA) += yama

  [...]
- obj-$(CONFIG_SECURITY_YAMA) += yama/
- These two lines integrate the security/yama directory into the kernel compilation process (if the related Kconfig option is enabled, of course).

LSM Code integration with the Kernel

- The major LSM framework code is contained in security/security.c
- This LSM core does the LSM framework initialization calling security_init() which loads the enabled supported Linux security modules in the order
  - Capability module
  - Minor LSMs
  - Major LSM
- The security_add_hooks() will register the specified LSM, E.g Yama LSM

```
static int __init yama_init(void)
{
        pr_info("Yama: becoming mindful.\n");
        security_add_hooks(yama_hooks, ARRAY_SIZE(yama_hooks), "yama");
        yama_init_sysctl();
        return 0;
}
```

security_add_hooks()

- The arguments passed to this function are the <LSM-name>_hooks array and its size, obtained with the ARRAY_SIZE() macro.
- This array is an array of security_hook_list structures and is defined in the same file, i.e. security/yama/yama_lsm.c for Yama:
- static struct security_hook_list yama_hooks[] __lsm_ro_after_init = {

```
    LSM_HOOK_INIT(ptrace_access_check, yama_ptrace_access_check),
    LSM_HOOK_INIT(ptrace_traceme, yama_ptrace_traceme),
    LSM_HOOK_INIT(task_prctl, yama_task_prctl),
    LSM_HOOK_INIT(task_free, yama_task_free),
};
```

- These security_hook_list structures are obtained with the LSM_HOOK_INIT() macro defined in include/linux/lsm_hooks.h.

Linking of hooks for stacking LSM

- This mechanism is used for every single hook defined by every single LSM integrated to the Linux kernel and enabled.
- The security_add_hooks() function is then expected to correctly chain all those security_hook_list structures.
- This way, we obtain one linked list per LSM and one linked list per hook provided by the LSM framework, depending on how we walk through those lists.
- In other word, it is these few structures, functions and macros which implement the stacking of LSM hooks, which is itself a consequence of the stacking of the security modules that define them.

Kernel calling LSM Hooks

- Kernel functions that contain LSM hooks, call the related LSM hooks wrapper functions defined in security/security.c
- These wrapper functions, in turn calls
  - call_int_hook() or
  - call_void_hook()
- For instance, yama_ptrace_traceme(), which corresponds to the ptrace_traceme() kernel function defined in kernel/ptrace.c:

```
static int ptrace_traceme(void)
{
   /*....*/
        ret = security_ptrace_traceme(current->parent);
  /*....*/
}
```

Kernel calling LSM Hooks

We see the call to security_ptrace_traceme(), defined like the following in security/security.c:

```
int security_ptrace_traceme(struct task_struct *parent)
{
        return call_int_hook(ptrace_traceme, 0, parent);
}
```

These call_int_hook() and call_void_hook() simply iterate through the linked list corresponding to the hook,

Call's the LSM hooks defined by the security modules that are enabled on the running system:

call_int_hook/call_void_hook macros

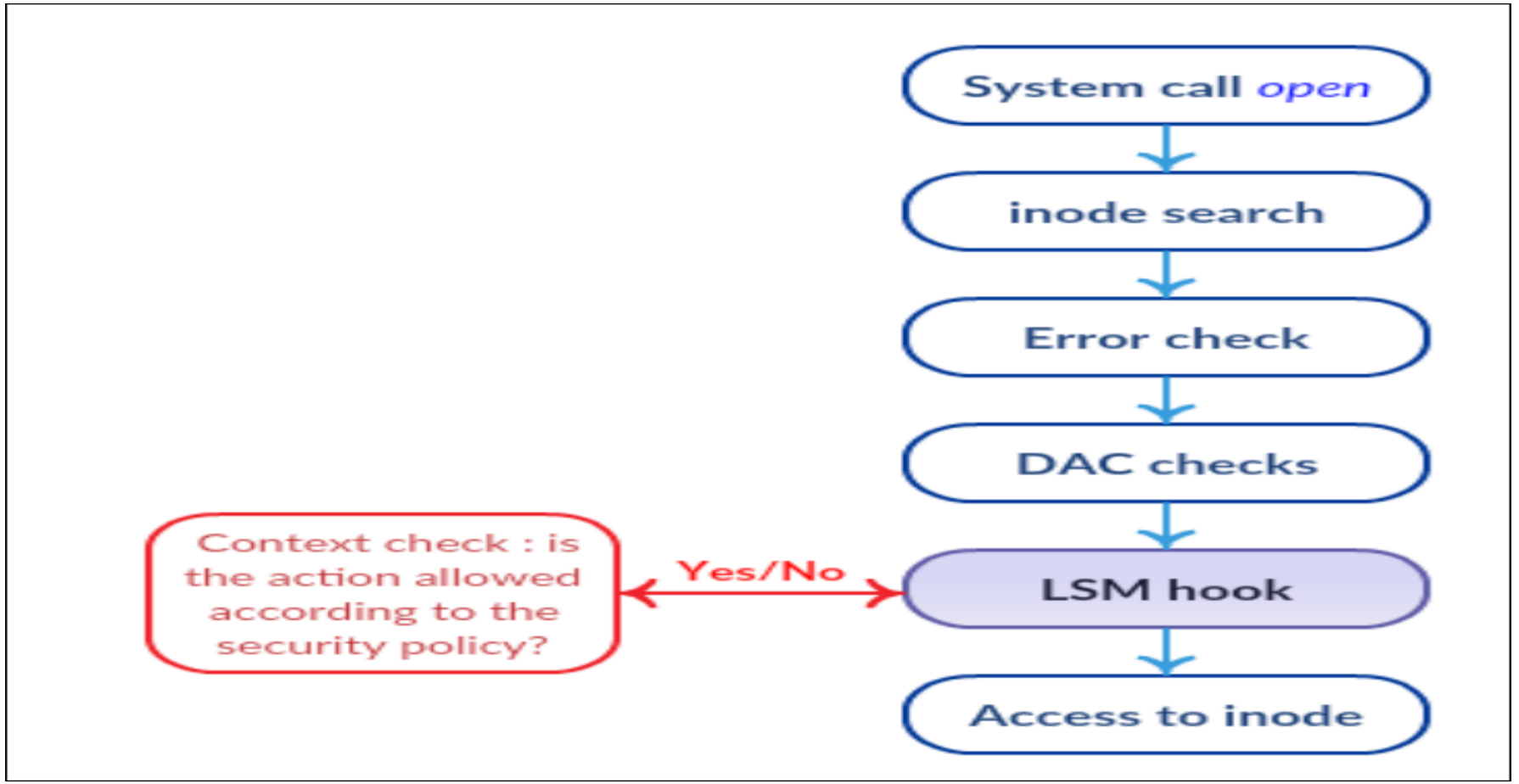
```
#define call_void_hook(FUNC, ...)                                       \
        do {                                                            \
                struct security_hook_list *P;                           \
                                                                        \
                hlist_for_each_entry(P, &security_hook_heads.FUNC, list) \
                        P->hook.FUNC(__VA_ARGS__);                      \
        } while (0)

#define call_int_hook(FUNC, IRC, ...) ({                                \
        int RC = IRC;                                                   \
        do {                                                            \
                struct security_hook_list *P;                           \
                                                                        \
                hlist_for_each_entry(P, &security_hook_heads.FUNC, list) { \
                        RC = P->hook.FUNC(__VA_ARGS__);                 \
                        if (RC != 0)                                    \
                                break;                                  \
                }                                                       \
        } while (0);                                                    \
        RC;                                                             \
})
```

- In the case of a hook returning an integer value, the iteration is interrupted as soon as one hook function returns a value different from 0, thus satisfying the “cannot override a denial” rule.

- By providing Linux with the standard API for policy enforcement, LSM ensures to enable the widespread deployment of security hardened systems

Flow of open() system call



Flow of open() system call

- The LSM framework integrated into the kernel provides each LSM with hooks on essential functions of the kernel. The diagram above shows coarse call flow of the “open()” system call
  - A process in the user space calls open() on the file path
  - The system call is dispatched and the path string is used to obtain a kernel file object and inode object.
  - If the parameters are incorrect, an error is returned
  - The normal “Discrete access control” (DAC) file permissions are checked. Does the current user have the permission to open the file is checked and if not, the system call is terminated and error is returned to user space.

Flow of open() system call

5) If DACs are satisfied, the LSM framework acts for each of the file_open hooks for the enabled LSMs. The system is terminated and error is returned to the user space if a single LSM hook returns an error.
6) Finally, if all security checks pass, the file is opened for the process and a new file descriptor is returned to the process in the user space.

LSM file system hooks

- The VFS layer defines three primary objects which encapsulates the interface that low level file systems are developed against
  - The super_block object
  - The file object
  - The inode object
- Each of these objects contains a set of operations that define the interface between the VFS and the actual file system
- This interface is a perfect place for LSM to mediate file system access.
- LSM uses the opaque security pointers defined in the kernel objects
  - super_block structure
  - file structure
  - inode structure

LSM super_block hooks

- The “super_block” structure is the kernel object representing the file system
- The structure is used while mounting and unmounting a file system or obtaining file system statistics
- LSM provides hooks to mediate the various actions on the super_block
- When mounting a filesystem, the kernel first validates the request by calling the sb_mount() super_block hook.
- While unmounting a filesystem, the super_block sb_umount() hook is called to check permissions to unmount the filesystem.
- The sb_remount() hook verifies the mount options
- The sb_statfs() hook checks permissions when a task attempts to obtain the filesystem statistics

LSM file hooks

- The “file” structure is the kernel object representing the open file
- It contains the “file_operation” structure which describes the operations which can be done to a file. For example, a file can be read from, written to , seek'd through, mapped into memory and so on.
- LSM provides a group of file hooks to mediate access to files.
- file_permission() hook can be used to re-validate read and write permissions at each file read and write.
- file_locks() hook can be used when using locks to synchronize multiple reader or writers, a task must pass the file_locks() hook permission before performing any locking operation on the file.
- file_ioctl/file_fcntl hooks can be used to miscellaneous file operations that come through ioctl(2) and fcntl(2).

LSM inode hooks

- The “inode” structure is the kernel object representing the kernel file objects such file, directory or symlinks
- LSM provides a group of hooks that mediate access to the fundamental kernel structure.
- The kernel's inode cache is populated by either the file lookup operations or the file system object creation operations
- A set of well-defined operations describes the actions taken on the inode, such as
  - create(), mkdir(), rmdir(), mknod(), rename(),
  - link(), unlink() , symlink(), readlink(), follow_link()
  - getattr(), setattr(), getxattr(), setxattr(), permissions()

LSM Task hooks

- The “task_struct” structure is the kernel object representing the kernel schedulable tasks. It contains basic task information such as user/group ID, resource limits, and scheduling policies and priorities
- LSM provides a group of task hooks that mediate a task's access to the basic task information
- task_alloc() hook is called to verify task can spawn children
- task_kill() hook is called when the task exits
- During the life of the task, some information may be changed such as call to setuid(2) system call, this in turn will call task_fix_setuid() hook

LSM IPC hooks

- The kernel provides the standard SysV IPC mechanins
  - Shared memory, semaphores and message queues
- LSM provides set of IPC hooks that mediate access to the kernel's IPC objects
- ipc_permission() hook check the IPC permissions
- msg_queue_msgrcv() hook check permission before the message is removed from the message queue
- shm_shmat() hook check permission before the shmat(2) to attach shared memory segment with the given permission to the data segment of the calling process.
- sem_semctl() hook check permission when a semaphore operation specified with the given command to be performed on the semaphore.

LSM Network hooks

- As networking is an important aspect of Linux and more importantly securing the system from network attacks, LSM provided the the extended security to this area of the kernel.
- Application layer access to networking is mediated via a series of socket-related hooks
- Additionally finer-grained hooks have been implemented for IPV4, Unix domain, netlink, infiniteband and SCTP protocols.
- Hooks provided for all the socket system calls:
  - bind(), connect(), listen(), accept(), sendmsg(), recvmsg(),
  - getsockname(), getpeername(), getsockopt(), etc
- Network data traverses the network stack in packets encapsulated by sk_buff structure, LSM provides opaque security field to the sk_buff so that security state can be managed across network layers on the per-packet basis.

LSM Module & System hooks

- The LSM framework would be incomplete if it didn't mediate loading and unloading kernel modules
- The LSM module loading hooks add permission checks providing the creation and initialization of loadable kernel modules
- LSM defines hooks for all security Key management operations
- LSM defines hooks for checking permission to change system time, allocating a new virtual memory mapping, accessing kernel message ring
- LSM hooks for Audit framework
- LSM hooks for using eBPF and programs functionalities through the eBPF system calls
- LSM defines a miscellaneous set of hooks to protect the remaining security sensitive actions that are not covered by the hooks discussed above.

Current LSM status

- As of kernel version 5.7, there are 9 LSMs
  - SELinux
  - SMACK
  - AppArmor
  - TOMOYO
  - Yama
  - LoadPin
  - SafeSetID
  - Lockdown
  - BPF

SELINUX

- First merged as part of 2.6, Selinux is the default MAC implementation on Redhat distribution
- SELinux consists of a Linux security module and set of trusted services for administration and secure system execution
- This SELinux mandatory protection system enables comprehensive control of all processes, so policy writers can exactly define the required accesses

SMACK

- SMACK like SELinux is an attribute based MAC implementation
- It was the second LSM development merged as part of 2.6.24 release
- SMACK was designed for embedded systems and to be simpler to administer and is the default MAC implementation in Automotive Grade Linux and Tizen
- It enforces additional restrictions on what subjects can access which objects, based on the labels attached to each of the subject and the object.
- SMACK uses extended attributes (xattrs) to store labels on filesystem objects and are stored in the extended security namespace

APPARMOR

- AppArmor is another MAC implementation, merged as part of 2.6.36 release and a default MAC implementation in Debian-based systems
- AppArmor is path based implementation rather than attribute based
- Policies based on paths can protect files on any file system since extended attributes are not required for storing security context information.
- Rules can also be specified for files that may not exist yet since the path can be stored

TOMOYO

- TOMOYO is, like AppArmor, another path-based MAC implementation and was first merged as part of Linux 2.6.30.
- The technique used to enforce MAC is called domain which is determined by the process execution history and each domain is represented by a concentration of all the previously executed path names

LoadPin

- LoadPin, merged in Linux 4.7, is a “minor” LSM
- LoadPin ensures all kernel-loaded files (modules, firmware, etc) all originate from the same filesystem, with the expectation that such a filesystem is backed by a read-only device such as dm-verity or CDROM.
- This is intended to simplify embedded systems that don’t need any of the kernel module signing infrastructure / checking if the system is configured to boot from read-only devices

Yama

- Yama, merged in Linux 3.4, is an LSM intended to collect system-wide DAC security restrictions that are not handled by the core kernel.
- It currently supports reducing the scope of the ptrace() system call so that a successful attack on one of a user's running processes cannot use ptrace to extract sensitive information from other processes running as the same user.

SafeSetID

- SafeSetID, merged in Linux 5.1, is an LSM used to restrict UID/GID transitions from a given UID/GID to only those approved by a system-wide whitelist.
- SafeSetID gates the setid family of syscalls to restrict UID/GID transitions from a given UID/GID to only those approved by a system-wide whitelist.
- These restrictions also prohibit the given UIDs/GIDs from obtaining auxiliary privileges associated with CAP_SET{U/G}ID, such as allowing a user to set up user namespace UID mappings.

Lockdown

- Lockdown LSM merged in Linux 5.4, implements a “lockdown” feature for the kernel. When lockdown is enabled, a kernel command-line parameter can be used to lockdown the kernel for integrity or confidentiality.
- When lockdown is set to integrity, features that allow userspace to modify the kernel are disabled such as
  - Unsigned module loading, access to /dev/{mem,kmem,port},
  - kexec of unsigned images, hibernation, direct PCI access,
  - Raw io port access, raw MSR access, modifying ACPI tables, unsafe module parameters, unsafe mmio, and debugfs access.
- When lockdown is set to confidentiality, along with integrity features,
  - Disabling of the features that allow userland to extract potentially confidential information from a running kernel such as /proc/kcore access,
  - Use of kprobes, use of bpf to read kernel RAM, unsafe use of perf, and use of tracefs.

Conclusion

- LSMs are not designed to prevent a process from being attacked.
- Good coding practices, configuration management, and memory safe languages are the tools for that.
- The protections provided by LSMs do, however, help protect your system from being hacked when an attacker exploits flaws in one of the running programs.
- They can be an important layer in any defense in depth strategy on Linux systems, and by understanding what protections they provide, you hopefully have a greater appreciation for what systems need to protect and how to implement those protections.

Thank you !!


THE
LINUX
FOUNDATION


THE
LINUX
FOUNDATION