CS 423: Operating Systems Design
MP4: Linux Security Modules

1 Goals and Overview

- Understand Linux Security Modules
- Understand the basic concepts behind mandatory access control
- Understand the basic concepts behind security IDs and opaque security contexts
- Understand the basic concepts behind extended file system attributes
- Add custom kernel configuration parameters and enable them upon compilation
- Design and Implement a simple Linux Security Module
- Design a Least Privilege Policy for `/usr/bin/passwd`
- Enforce the implemented policy using the implemented LSM

2 Introduction

The Linux Security Module (LSM) project grew out of a discussion initiated by the NSA when they presented their work on Security Enhanced Linux (SELinux) at the Linux Kernel Summit in 2001. The goal of the LSM project is to provide a framework for general access-control without the need to modify the main kernel code. By default, the Linux kernel provides support for discretionary access control, and prior to LSM, lacked support for more general access control mechanisms. The LSM framework has allowed developers to add support for various security models without the need for changes to the core kernel code. The currently accepted security modules in the mainstream kernel are `AppArmor, SELinux, Smack, TOMOYO Linux`, and `Yama`.

In order to allow for module stacking, the security modules are separated into *major* modules and *minor* modules. There can only be one major security module running in a given system, while

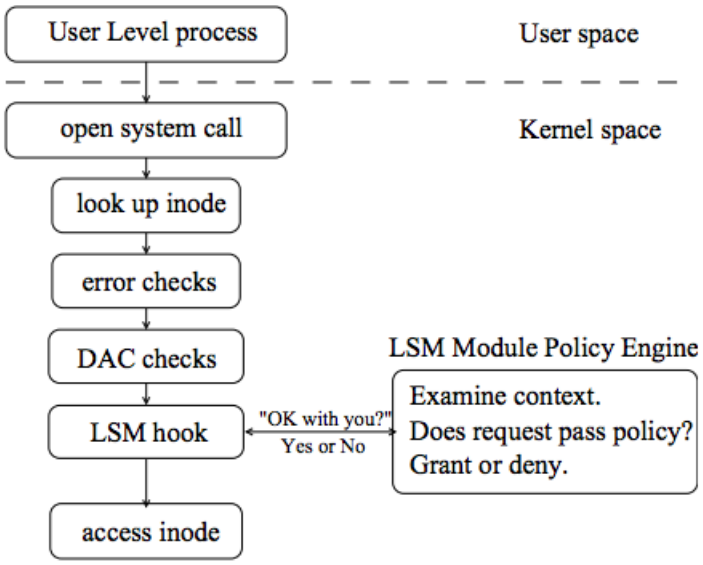

Figure 1: LSM Hook Architecture Example [5]

minor modules can be stacked to provide different security features. Additionally, minor modules are not allowed to access to the opaque security fields provided by the LSM framework (more on this later in the document). In this machine problem, we will be developing a major security module.

The LSM framework works by introducing hooks into a wide variety of kernel functionalities. Generally, the access control checks occur in the following order: after performing error checking, the kernel consults the discretionary access control mechanism, then calls the hooks for the minor modules if any are present, followed by the hooks of the major security module in place at the time. The full list of the hooks exposed to security modules, along with their description is located in `include/linux/lsm_hooks.h`

Figure 1 shows the LSM hook architecture for a sample `open` system call. Note that security modules, unlike the modules we have previously developed, are not loadable at runtime (the term "module" in LSM is somewhat of a misnomer). Security modules must be compiled alongside the kernel and enabled through the appropriate configuration switches. Default modules can be configured at build-time using `CONFIG_DEFAULT_SECURITY` but can also be overridden at boot-time using the `security=...` kernel boot command line parameter.

3 Developmental Setup

In this assignment, you will again work on the provided Virtual Machine and you will develop your security module for the `linux-4.4` kernel that you compiled as part of MP0. Again, you will have

full access and control of your Virtual Machine, you will be able to turn it on, and off using the VMWare vSphere Console. Inside your Virtual Machine you are free to install any editor or software like Gedit, Emacs and Vim. *To avoid any interference in your VM from the MP3 grading, make sure to reboot your VM immediately when you start this assignment.*

Development in this assignment is different from previous machine problems. Security modules must be compiled with the kernel at build-time, and appropriate configuration parameters should be enabled for them to work correctly. **Before starting this assignment, please take a snapshot of your vm in working condition**. The development you will do in this machine problem will affect your kernel's boot procedure and bugs in your code are likely to prevent your machine from booting and/or break the filesystem. Furthermore, if you have access to a more powerful machine on which you would like to carry on the compilation of the kernel and the production of the `.deb` packages, you are allowed to do that as long as a copy of your code ends up on your virtual machine and the VM is running the correct kernel at the time of the grading.

Finally, you are encouraged to discuss design ideas and bugs in Piazza. Piazza is a great tool for collective learning. However, please refrain from posting large amounts of code. Two or three lines of code are fine. High-level pseudo-code is also fine.

4 Problem Description

In this MP, we will develop a Linux security module that implements a mandatory access control policy for file accesses in the Linux environment. The module will only focus on programs labeled by the `target` label, and provide minimum access control for the remaining program. The goal of this approach is to ensure proper functioning of the virtual machine since our access control policy does not cover all the possible cases. Our module will be embedded into the kernel at build-time and we will select it as the default major security module at boot time. Additionally, we will inspect the user-space program `/usr/bin/passwd` and generate the Least Privilege Policy for it. We will then label our file system accordingly and verify that our security module can enforce that policy.

Figure 2 shows an overview of the process that happens in our security module when a program attempts to access an inode. In order to be able to enforce an access control policy, our module will require security labels on the inodes and the tasks in the system. For the inodes, we will make use of the filesystem's extended attributes (or `xattr`). Extended attributes provide us with a way to associate our file with metadata that is not interpreted by the filesystem. Instead the metadata will be interpreted by our custom security module. It is common practice for linux security modules to use extended attributes under the prefix `security`. In this machine problem, each inode that we care about will be assigned an attribute value under the name `security.mp4`.

As for the linux tasks, we will make use of the opaque security fields that are provided by the

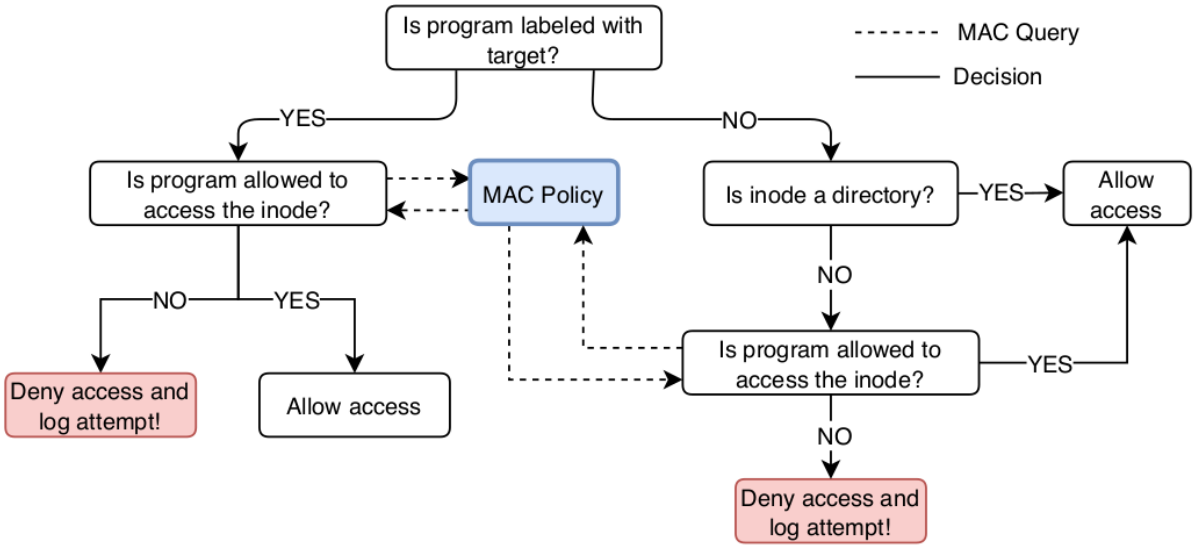

Figure 2: Architecture of MAC checks in MP4

kernel. Opaque security fields (or from now on, security blobs) are `void *` pointers available in the credentials structure (`struct cred *`) associated with each `task_struct` created by the kernel. In this MP, these blobs will contain the security label of the task being run, which we will obtain by examining the security attribute of the binary file used to launch the process (in other cases, for example a `fork`, the process will inherit its parent's security blob).

As previously mentioned, our module specifically enforces access control for the programs that have been labeled with the `target` label. **For those programs that are not labeled as `target`, our module will allow them full access to directories (regardless of the directories' security labels), and will allow them read-only access to files that have been assigned one of our custom labels.** The labels we will use along with their semantics are shown in Figure 3.

5 Implementation Overview

5.1 Compiling and Configuring the kernel tree

1. We will start by installing the necessary packages needed for this MP:

   ```
   sudo apt install -y libncurses5-dev attr
   ```

2. Next we need to add configuration options for our security module in the kernel's `Kconfig` environment. Start by creating a directory for the module in the kernel's security directory. From the root of the kernel source tree, use

```
/* mp4 labels along with their semantics */
#define MP4_NO_ACCESS 0       /* may not be accessed by target,
                               * but may by everyone other */

#define MP4_READ_OBJ 1        /* object may be read by anyone */

#define MP4_READ_WRITE 2      /* object may read/written/appended by the target,
                               * but can only be read by others */

#define MP4_WRITE_OBJ 3       /* object may be written/appended by the target,
                               * but not read, and only read by others */

#define MP4_EXEC_OBJ 4        /* object may be read and executed by all */

/* NOTE: FOR DIRECTORIES, ONLY CHECK ACCESS FOR THE TARGET SID, ALL OTHER NON
 * TARGET PROCESSES SHOULD DEFAULT TO THE LINUX REGULAR ACCESS CONTROL
 */
#define MP4_READ_DIR 5        /* for directories that can be read/exec/access
                               * by all */

#define MP4_RW_DIR 6          /* for directory that may be modified by the target
                               * program */
```

Figure 3: The labels that we will use for our security module.

`mkdir -p security/mp4`

In the newly created `mp4` directory, create a file named `Kconfig`. This file will define the configuration entries for our module as follows

```
1  config SECURITY_MP4_LSM
2          bool "CS423 machine problem 4 support"
3          depends on NET
4          depends on SECURITY
5          select NETLABEL
6          select SECURITY_NETWORK
7          default n
8          help
9            This selects the cs423 machine problem 4 security lsm to be
10           compiled with the kernel.
11           If you are unsure how to answer this question, answer N.
```

3. Next, under the `security/` directory, modify the `Makefile` and the `Kconfig` files to read from our module's appropriate files. Add the following to line 125 of `Kconfig`

   `source security/mp4/Kconfig`

   Also, modify the `Makefile` to include the mp4 module by adding the following at the appropriate

location:

`subdir-$(CONFIG_SECURITY_MP4_LSM) += mp4`

also add this at the appropriate location:

`obj-$(CONFIG_SECURITY_MP4_LSM) += mp4/`

4. You can now compile the kernel in the same way we did for MP0. When you get to the configuration phase, you will be prompted whether you want to enable `CONFIG_SECURITY_MP4_LSM`, answer this with `yes` and continue with your compilation.

   NOTE: If you are not making use of debugging software, you might want to compile your kernel without the DEBUG symbols to speed up the process of generating the Debian packages. To do that, you can use `make menuconfig` which will provide you with an in-shell UI that you can use to configure the kernel build. Navigate to the `DEBUG_INFO` entry in the configuration and switch it off. You can also check and change the value of our custom configuration entry from this menu.

   Additionally, you do not need to clean the kernel build every time you make some changes to your source code, it is better to build incrementally. When actively developing, you can compile the kernel without producing the `.deb` packages using `make -j<num_proc>`. When you are ready to generate your packages, you can use `make -j<num_procs> bindeb-pkg`.

5. Finally, you need to add your security module as the default only major LSM to run on your machine. To do that, we will edit the `grub` configuration and make the change persistent (the option will be ignored for other kernel build that do not support the module). Open the file `/etc/default/grub` with your favorite editor and change the line

   `GRUB_CMDLINE_LINUX_DEFAULT="quiet nosplash"`

   to

   `GRUB_CMDLINE_LINUX_DEFAULT="quiet nosplash security=mp4"`

   And then update grub using `sudo update-grub`.

5.2 Implementing and Testing the MP4 LSM

Just like any other module, Linux security modules must have an initialization function and must register hooks with the kernel's security system. We have provided you with the functions that perform these operations in the file `mp4.c`. In addition, `mp4_given.h` provides you with the definition of an mp4 custom security structure along with some additional helper functions that you might find useful. We recommend that you use `pr_info`, `pr_debug`, and `pr_err` for logging. The line

at the top of `mp4.c` ensures that all messages printed using these functions will get prefixed with `cs423_mp4`. This will make it easier for you to find your log messages in the kernel log dumps.

In this MP, we will be implementing six hooks into the kernel. We have provided you with the code that registers these hooks with the system along with their empty skeletons in `mp4.c`. These hooks are:

1. `mp4_cred_alloc_blank`
2. `mp4_cred_prepare`
3. `mp4_cred_free`
4. `mp4_bprm_set_creds`
5. `mp4_inode_init_security`
6. `mp4_inode_permission`

1. Start by implementing `mp4_cred_alloc_blank`, `mp4_cred_prepare`, and `mp4_cred_free`. These hooks are used to allocate (and free) memory for our security blobs. Your structures should always be initialized to the `MP4_NO_ACCESS` label.

2. Next you should implement the `mp4_bprm_set_creds` hook. This hook is responsible for setting the credentials context (and thus our security blob) for each process that is launched from a given binary file. Your code should read the `xattr` value of the inode used to create the process, and if that labels reads `MP4_TARGET_SID`, you should set the created task's blob to `MP4_TARGET_SID` as well.

   For your convenience, we have provided you with the function
   `static inline int __cred_ctx_to_sid(const char *cred_ctx)`
   that will convert an `xattr` text value into the appropriate sid value.

   For obtaining the `xattr` of an inode, you might find it useful to look at the source code of the filesystem you are using and at other drivers/modules in the kernel code. Note that when dealing with `struct dentry*`, the kernel keeps reference counts for each pointer you use, so you MUST always put a `dentry` structure back using the call to `dput(struct dentry *)`. Not doing this will induce errors in your filesystem. While modern filesystem can detect these errors and remove dangling `dentries` at boot time, this will significantly increase your boot time and might break your filesystem.

3. Next you should implement `mp4_inode_init_security`. This hook is responsible for setting the `xattr` of a newly created inode. This value will depend on whether the task that creates this

inode has the `target` sid or not. For those inodes that were created by a target process, they should always be labeled with the `read-write` attribute. For all other inodes, you should not set any `xattr` value.

4. Finally, you should implement the `mp4_inode_permission` hook. This hook implements our main mandatory access control logic, and should be a direct translation of the label semantics and operation shown in Figures 2 and 3. In order to speed up the boot process, your code **MUST** first obtain the path of the inode being checked, and then use the helper function `mp4_should_skip_path` provided to you in `mp4_given.h` to skip over certain paths heavily used during boot time, and that we will not touch throughout our implementation. Hint: In order to obtain the path of an inode, you can use its dentry.

   The macro `current_cred()` can help you obtain the security blob of the current task. Additionally, you can find the definition of the filesystem operation masks in `linux/fs.h`.

   Furthermore, your access control logic **MUST** log all failed access attempts to the kernel logs. However, be careful not to overload the boot process with printouts. If you must print debug messages, you may find the function `int printk_ratelimit(void)` useful.

   In order to avoid breaking your system due to incorrect permission checks during boot time, we recommend that your initial access control checks would perform the logic but always grant access and log the requests that should have been denied. Then you can test your code's correctness by looking at the kernel logs and making sure that printed message are correct. After doing that, you can then return the appropriate values in your code and test their operation.

5.3 Testing Your Code

In order to test your code, pick a harmless application (e.g., `cat` or `vim`) and label its binary as your target application. We recommend that you follow the following procedure.

1. Create two scripts called `test.perm` and `test.perm.unload`, the first script will contain the commands that set the security attribute value for files in your system, while the second will reverse all the effects created by the first script. We strongly recommend that you always remove all of the attributes you set during a testing session in order to ensure that your system can boot successfully the next time you reboot.

2. If you would like to provide `cat` with read-only access to the file `/home/NETID/file.txt`, we first start by labeling `cat`'s binary as the target, and then providing cat with directory access rights to `/home` and `/home/NETID`. And finally we would label `/home/NETID/file.txt` with the `read-only` attribute. Your `test.perm` should look something like

```
setfattr -n security.mp4 -v target /bin/cat
setfattr -n security.mp4 -v dir /home
setfattr -n security.mp4 -v dir /home/NETID
setfattr -n security.mp4 -v read-only /home/NETID/file.txt
```

3. You can then source your script for the changes to take effect using `source test.perm`

4. Test `cat`'s access to the file by trying to read it. You can then change the permissions on the file and then try again to make sure your module is working for all the cases.

5. Don't forget to source your unloading script when you're done `source test.perm.unload`

5.4 Obtaining the Least Privilege Policy for `/usr/bin/passwd`

In the final piece of the MP, you will generate the Least Privilege Policy for `/usr/bin/passwd` and then make sure that your module can enforce it. You will generate two scripts, `passwd.perm` and `passwd.perm.unload` and submit them as part of your MP submission. In order to obtain the files and the permissions that `/usr/bin/passwd` requires to run correctly, you should make use of `strace` tool. Start by creating a dummy user on your vm that you can use to test things out. PLEASE DO NOT TRY TO CHANGE THE PASSWORD FOR YOUR USER ACCOUNT; your accounts are configured to use the University's Active Directory servers.

- Create a dummy user and change its password. Use `strace` to profile `/usr/bin/passwd`'s run and look at the files that it requests access to.

- Look into the report generated by `strace` and figure out the files and the permissions that `/usr/bin/passwd` requests access to. We recommend that you use a text manipulation tool such as `sed` or `awk` to speed up this process.

- Generate the least privilege policy for `/usr/bin/passwd` and store it in `passwd.perm`. Note that your script MUST always start by setting the target label on `passwd`'s binary.

- Load the policy and test your module's enforcement.

6 Software Engineering

Your code should include comments where appropriate. It is not a good idea to repeat what the function does using pseudo-code, but instead, provide a high-level overview of the function including any preconditions and post- conditions of the algorithm. Some functions might have as few as one

line comments, while some others might have a longer paragraph. Also, your code must be split into small functions, even if these functions contain no parameters. This is a common situation in kernel modules because most of the variables are declared as global, including but not limited to data structures, state variables, locks, timers and threads. An important problem in kernel code readability is to know if a function holds the lock for a data structure or not, different conventions are usually used. A common convention is to start the function with the character '_' if the function does not hold the lock of a data structure. In kernel coding, performance is a very important issue, usually the code uses macros and preprocessor commands extensively proper use of macros and identifying possible situations where they should be used is important in kernel programming. Finally, in kernel programming, the use of the `goto` statement is a common practice. A good example of this, is the implementation of the Linux scheduler function schedule(). In this case, the use of the `goto` statement improves readability and/or performance. "Spaghetti code" is never a good practice!

7 Submission Instructions

Push your all your relavant code to your assigned github repo. Please make sure you have the policy files that you generated for `passwd`, namely `passwd.perm` and `passwd.perm.unload`.

8 Grading Criteria

Criterion	
The VM is running the correct kernel and the mp4 security module is loaded	
Correct implementation of security blobs allocation and freeing	
Correct implementation of the transfer of labels from the binary to the `task_struct`	
Correct assignment for the labels of inodes created by a target program	
Correct implementation of the access control enforcement	
Obtained the least privilege policy for `/usr/bin/passwd`	
Your module correctly enforces the policy for `/usr/bin/passwd`	
Document briefly describing your implementation as in Section 8	

References

[1] The linux kernel source tree. Available at `https://elixir.bootlin.com/linux/v4.4/source`.

[2] Linux security module usage. Available at https://01.org/linuxgraphics/gfx-docs/drm/admin-guide/LSM/index.html.

[3] David Howells. Credentials in linux. Available at https://www.kernel.org/doc/Documentation/security/credentials.txt.

[4] Greg Kroah-Hartment. Using the kernel security module interface. OUTDATED but provides useful information. Available at https://www.linuxjournal.com/article/6279.

[5] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, 2002.