Linux Kernel Programming

Lab 07: Virtual File System

AUTHOR
Redha Gouicem and Julien Sopena

Task 1: Patching the kernel

In this lab, you will need a slightly modified version of the kernel to export some variables. Instead of directly changing the sources, you will apply the patch provided to you in `sources/export_dentry_hashtable.patch`.

Question 1

Read the patch to understand what it does.

Apply the patch with the `patch` command from the root of your kernel source tree. Note that you will need the `-p1` option to tell `patch` to ignore the first level in the paths of the patch.

> **Tip**
>
> If building the kernel takes a lot of time on your machine, **don't do it yet**! Wait for the next question!

Question 2

For this lab, you will also need to access the variable `d_hash_shift`. Find where it is located in the kernel source. Can you use it in a module? If not, make the necessary changes to be able to.

Question 3

In order to be able to easily apply these changes in the future, generate a new patch that contains the changes of the patch provided to you as well as your modifications.

You can generate a patch with the `diff` command. Make sure to use the unified format and to compare the whole source tree. You can check the man page of the command as well as the patch provided earlier for some tips.

Task 2: A not so private cache

Question 1

The *dentry cache* is a hash table that stores directory entries. By having a look at the kernel sources, find out the size of the hash table, *i.e., the number of lists in it*. More specifically, you can check out the `d_hash()` function.

Question 2

Implement a module called *weasel* that prints the address of the *dentry cache* hash table as well as its size, *i.e., the number of buckets of the hash table*.

Question 3

Modify your module to print the total number of entries in the table when you load it, as well as the length of the longest bucket.

What are your thoughts about the size of this cache, and the number of entries it contains? To help you have a better idea of this cache's size, print the size of a *dentry* as well as the memory footprint of the *dentry cache*.

You can also fill up the cache with valid entries by accessing all the files in the system by running:

```
find / -type f -print0 | xargs -0 stat > /dev/null
```

You can also try to access random file names to create *negative dentries*.

> **Caution**
>
> The lists used in the hash table of the *dentry cache* are not `struct list_head` s. They are a slightly different kind of list with a particular feature that you **must** use to access the lists safely! Have a look at their definition in the kernel sources and how to use them properly.

Question 4

We now want to be able to query the state of the *dentry cache* without having to load and unload the module. To do so, you will use the *procfs*.

Modify your module so that it creates the directory `/proc/weasel`, as well as a file `/proc/weasel/whoami` that prints `"I'm a weasel!"` when read. You can use the function `proc_mkdir()`, `proc_create()` and `remove_proc_entry()`.

> **Tip**
>
> You will need to declare a `struct proc_ops` for your *procfs* file. ALthough the name of the members differs, you can use it in the same way as a `struct file_operations`.

Question 5

In order to export the *dentry cache* content to user space, implement a *procfs* file `/proc/weasel/dcache` that lists the content of the *dentry cache*. You can implement this with a simplified version of *seq files*. Instead of implementing a `struct seq_operations`, you can just use the `single_open()` function in your open function in the `struct proc_ops` and set the other function pointers properly. You can find multiple examples in the kernel sources.

> **Tip**
>
> To get the full path of a *dentry*, you can look up exported functions in `fs/d_path.c`.

Question 6

In the terminal of the VM, try to execute a non-existent program, *e.g., foo* and look it up in your `/proc/weasel/dcache` file. From this output, infer the content of your `$PATH` environment variable. Why are these errors recorded in the *dentry cache*?

Question 7

Sometimes, when a user tries to authenticate themselves with a command such as `su`, they might type their password a second time directly in case of failure. This leads the shell to interpret this as a command.

Add a new *procfs* file in your module, `/proc/weasel/pwd`, that only display the list of commands that were not found in the `$PATH`.

Task 3: Rootkit – Hide a process (naive)

> **Note**
>
> The instructions in this exercise are not too detailed **on purpose**! We only give you the minimal knowledge and all the keywords you need to look up what to do exactly in the kernel sources.

In this task, you will implement a module that hides a process from a user using commands such as `ps`. The PID of the hidden process can be passed as a module parameter.

Commands such as `ps` use the information about processes exported in `/proc`. Each process has a directory named after its PID created there, exporting data about the process. Thus, hiding a process from this family of programs requires to hide the presence of this directory in the *procfs*.

As you should know, the Virtual File System (VFS) acts as an abstraction layer between the generic file API and concrete file system implementations. It manipulates different types of objects: *files*, *dentries*, *inodes* and *superblocks*.

In this task, you will modify the functions implemented by the *proc* file system in order to hide a process.

Question 1

Implement the skeleton of your module that takes as a parameter the PID of the process to hide.

Question 2

In order to hide the directory that corresponds to this process, you need to override the `struct file_operations` of the `/proc` directory. More specifically, you need to override the function that lists the content of this directory to avoid showing the hidden process.

First of all, you need the `struct file` associated with the `/proc` directory. You can get it with the `filp_open()` function, which is the kernel space version of the `open` system call. Don't forget to close the file when you're done with it!

With this `struct file`, you will be able to override the *file operations* for `/proc`.

Question 3

Now, you need to hide the PID passed as a module parameter from programs querying the content of `/proc`. The `iterate_shared` member of the *file operations* is the function called when listing the content of a directory, *e.g., when calling* `ls`. It takes a `struct dir_context *` in parameter that defines how each entry of the directory is recorded. More specifically, this context object contains a function pointer of type `filldir_t` that is called for each entry in the directory to fill the list of entries in the directory.

Since we want to preserve the normal behavior of the *procfs* for all PIDs except for the one we are hiding, we can replace the `iterate_shared` function used for `/proc` so that it calls the default `filldir_t` function for all PIDs, and nothing for the PID we are hiding.

Implement a custom `iterate_shared` function as well as a custom `filldir_t` function. Then, replace the default behavior of the *procfs* with the one hiding the specified PID.

> **Important**
>
> Be careful! You need to revert the *procfs* to its normal behavior when unloading the module!