VFS-based File Operations: encrypted files

ESTR 3102 - Linux Kernel Hacking Assignment 2

Abstract

In the workshop, we learn about the VFS layer of Linux kernel. This assignment is a golden chance for you to understand the merit of the VFS layer: you can write codes that are independent of the underlying file systems.

1 Task Overview

In this assignment, you have the chance to hijack the following VFS system calls.

Syscall	Kernel source
`open()`	`fs/open.c`
`close()`	`fs/open.c`
`read()`	`fs/read_write.c`
`write()`	`fs/read_write.c`

Important. The aim of this assignment is to practise VFS-layer hacking, rather than secure system implementation. Therefore, the security practice of this assignment will be of the least concern.

You are asked to modify the Linux kernel such that it allows a process to encrypt and decrypt the file content when it invokes the `write()` and the `read()` system calls, respectively.

1.1 Flow

1. A process opens a file using a modified version of the `open()` system call called `eopen()`. You have to add this new system call to the Linux kernel with the following prototype:

```
long eopen(const char *ekey, const int elen, const char *filename,
int flags, int mode);
```

where,

- "`ekey`" is the encryption key for the opening file. Remember, it is not a system-wide encryption key, instead it is a per-file encryption key. Let us assume that the key contains ASCII characters only. It ends with the terminator "`\0`".
- "`elen`" is the length of the encryption key, in terms of bytes. You can assume that the maximum length of "`ekey`" is 255 bytes, not including the NULL terminator.
- Other than "`ekey`" and "`elen`", the remaining parameters of the new system call are the same as those you can find in the `open()` system call.
- `eopen()` should first check if the `ekey` is a non-empty, NULL-terminated string. If not, `eopen()` returns the error `EACCES` and the system call ends.
- Next, `eopen()` should memorize the `ekey` in the kernel space and then continues with the `open()` system call.
- No matter the `open()` system call returns successfully or not, the `eopen()` system call returns the return value from the `open()` system call.
- Suppose a process has opened a file using `eopen()`. If there is another process trying to open the same file again with `eopen()`, the `eopen()` calls from the latter process will receive the error `EACCES`.
- It is assumed that while a file is opened by `eopen()`, before it is closed, no processes would open the same file using the `open()` system call or the `eopen` system call.

2. After the `eopen()` has been invoked successfully, the protection over the opened file is on.
   - For each invocation of the `write()` system call, the data written to the file must be encrypted.
   - For each invocation of the `read()` system call, the data returned to the process must be decrypted.
3. The encryption protection is turned off whenever the file is closed or the calling process terminates.

1.2 Encryption and decryption

The recommended encryption function is to use the XOR operation.

- Let $K = \{K_0, K_1, \ldots, K_{n-1}\}$ be an encryption key of n bytes.

- Let $M = \{M_0, M_1, \ldots, M_{m-1}\}$ be a message (or file content) of m bytes.
- Both the encryption and decryption functions are the same, and produce an output $C = \{C_0, C_1, \ldots, C_{m-1}\}$, where the lengths of both C and M are the same, m bytes. The function is given as follows.

  $$C_i = M_i \oplus K_{i \ \% \ n}$$

  where

  - $\oplus$ means the bitwise XOR operator.
  - In both encryption and decryption functions, C represents the encrypted message and the decrypted message, respectively.

It is the property of XOR that:

$$(A \oplus B) \oplus B \ = \ A.$$

Therefore, if you encrypt the message M with key K using the above function, then you can always decrypt using the same function and get back the original message using the correct key K.

1.3 Assumptions and requirements

- You can assume that only regular files will be passed to the `eopen()` system call.
- For the `write()` system call, you cannot change the content of the buffer passed from the userspace. This is a **must** for every decent kernel programmer.
- You have to support both the sequential and the random access modes.
- It is allowed that multiple processes call *eopen()* over a set of distinct files at the same time.
- A process can call *eopen()* multiple times over a set of distinct files.
- Every file opened through *eopen()* will be properly closed by the `close()` system call.

1.4 Utility programs

You have to write utility programs that works as follows:

1. Encoder: `eopen()` → `read()` from stdin and `write()` to target file → `close()` when the stdin stream ends.

2. Decoder: `eopen()` → `read()` from target file and `write()` to the stdout → `close()` when the end of the target file is reached.

1.5 General requirements

- You must realize the task by hacking the Linux kernel. The required system is Gentoo with kernel version ≥ 4.0.
- Please use one CPU for this assignment since there may be locks implemented in the VFS layer.
- Using which file systems in this assignment is not a concern because we modify the VFS layer.
- We will always work on the same partition when we grade your assignment.

1.6 Milestones

- 3 marks: adding system calls and writing the utility programs.
- 3 marks: single process and single opened file.
- 3 marks: multiple processes and multiple opened file, using different encryption keys.

2 Submission Guideline

For the submission of the assignment, please send your work to:

`tywong@cse.cuhk.edu.hk`

Remember, you have to submit all the user programs and all the modified kernel codes (but, never the entire kernel source tree).

Deadline: 16:30, November 27, 2015 (Fri).

Happy Hacking.