CSE509 Project for Fall 2011

Encryption/Decryption of Files using System Call Interception

by

Piyush Kansal and Matthew Culver

Graduate Students, Computer Science

Stony Brook University, NY, USA

under guidance of

Prof. R. Sekar

Professor, Computer Science

Stony Brook University, NY, USA

TOPICS:

1. Motivation

2. Overview of the approach

3. Platform

4. Addressing multiple aspects of the project

5. Applications

6. References

1. MOTIVATION:

Because of ever-growing privacy and cyber security concerns, the ability to easily encrypt and decrypt files has become more and more desirable over the last several years. The purpose of this project is to provide a means for allowing a user or application the capability to encrypt files at will. The encryption process itself is accomplished via an **extension** over current file-handling primitives.

During the implementation of this project, we looked at things from a number of different perspectives and took many important issues into account, including *usability, security, performance, efficient usage of system resources, scalability, maintainability, and extendibility*. Our implementation addresses these important factors in various ways, as will be described later in this report.

2. OVERVIEW OF THE APPROACH:

The encryption capability involves the interception of system calls related to the file operations open(), read(), write(), close(), and lseek(). A new flag called O_ECRYPT was introduced in the local copy of *fcntl.h*, and this flag can be used normally with other system defined flags. For example:

open (<file-name>, O_CREAT | ***O_ECRYPT***);

Using a library interceptor received from *SecLab* [1], we intercepted the five system calls mentioned above and implemented the desired functionality in *syscall_handler_pre()* and *syscall_handler_post()* functions. For example, suppose open() is called. This system call is intercepted in syscall_handler_pre() and the status of O_ECRYPT is checked. If O_ECRYPT has been set in the parameters, we set up the required data structures for encryption in the program and then modify the original parameters to remove the flag O_ECRYPT, enabling the original

open() system call to proceed as normal.

The user can make any required changes to file data by making calls to lseek(), read(), and write() **any number of times** and **in any order**. Internal data structures will keep track of these changes and will temporarily decrypt the file. When the file is closed, it is automatically re-encrypted.

The algorithm *libmcrypt* is used for the actual encryption and decryption step, and the overall encryption is accomplished by way of a stream cipher. We decided for our purposes that the advantages of using a stream cipher over a block cipher, including increased memory efficiency for handling later modifications as well as a lack of susceptibility to frequency analysis attacks, outweighed the downside of potentially having to decrypt and re-encrypt the entire message each time a user tries to modify the encrypted code.

3. PLATFORM:

The platform we have chosen for development is Ubuntu 11.04.

4. ADDRESSING MULTIPLE ASPECTS OF THE PROJECT:

As previously mentioned, a fundamental goal of this project is to address the issues of usability, security, performance, efficient usage of system resources, scalability, maintainability, and extendibility. We will now discuss how we took these issues into account.

a) Usability

It is important that a user feels ***comfortable implementing encryption primitives***. The flag O_ECRYPT looks like and is used in a way that is similar to existing ones such as O_CREAT and O_RDWR, and this will hopefully provide some familiarity for the user. Applications such as internet browsers and office productivity suites would just need to provide an option to encrypt files (via their existing dialog boxes) to the user and can in this way set the O_ECRYPT flag while making a call to libc.

Also, the user should be ***able to randomly seek the file*** for reading and writing any number of times, and thus the encryption is designed so that it does not hinder these actions in any way. We have enabled this versatility by allocating memory during open() in syscall_handler_post() and decrypting the data of the whole file within the memory; thus, any further lseek(), read(), and write() operations are directly made to the memory location instead of to the actual file. Finally, during the close operation, in syscall_handler_pre() we encrypt the data in the memory and dump it in the file and finally free that memory, and then close the file in

syscall_handler_post().

Enabling this capability was a logical choice given the stream cipher implementation as well as the assumption that a user is more likely to want to make *multiple modifications to a single file than make few modifications to many files* (if the opposite were true, then a different implementation that uses a block cipher and only encrypts or decrypts parts of a single file might have made more sense).

In addition, we have provided a feature that allows a user to create a file with some given name and then later **rename the file** without losing any of the encrypted data (user can still perform reads and writes). This is made possible through an implementation that uses the **Inode number** for identification, which we are storing in the Keyring.

b) Security

As the professor suggested, the above approach can be vulnerable if the pages of the process are swapped out. We first tried using mlockall() to lock all of the pages but, due to a known bug in gcc's implementation of malloc [3], the use of memset yielded a SIGSEGV error response. So we instead decided to use **mlock() to lock the required buffers**. The buffers we locked using mlock() include the buffer in which we are storing the file data as well as the buffer in which we are storing the encryption key, random block, username and inode number.

To address the issue of storing and accessing the encryption key, we have taken the professor's suggestion and made use of **GNOME Keyring** to take care of key management. The provided primitives are just used to store and access the key. Short of somehow gaining administrator privileges, an attacker should have a very difficult time accessing the information stored by the Keyring; in essence, we are relying on the security of the underlying system itself for this part of the implementation.

Another potential issue is the possibility that, while creating the encrypted file, a user might mistakenly mark the file permissions to world readable or writable. Even if a file is already encrypted, an attacker with write permission can always alter the ciphertext in a way that causes the entire message to become corrupted. We have addressed this possibility by **masking i/p permissions** in such a way that only the owner of the file has the required access.

c) Performance

Our original intention was to implement random lseek(), read(), and write() in the encryption itself by starting to decrypt the text from the beginning of the file up to the point that the user wants to read/write, and then returning the requested range to the user. This was found to be a poor choice from a performance point of view because it complicates matters whenever the user attempts to make additional changes that are different from the original change. Next,

we made use of memory-mapped I/O. This was found to be an improvement from a performance standpoint over the previous approach but, for small files (i.e. less than a page size), it causes unnecessary **internal fragmentation**. This problem is exacerbated in 64-bit machines, where page size is much larger than it is for 32-bit machines.

In the end, we decided to use a **balanced approach** where we still use memory but only allocate bytes equal to the file size (BUFSIZ if the file is initially empty). If the file grows, we realloc() more memory in chunks of size BUFSIZ. This way, although memory is reallocated in size-BUFSIZ chunks, there is no internal fragmentation problem because it is requested from the heap and sent back as soon as the file is closed.

d) Space Efficiency

An important goal of this project was to produce an implementation that makes efficient use of system resources. One way that we did this was by using a stream cipher instead of a block cipher, since block ciphers would have potentially wasted disk space during modifications.

e) Scalability

We have made the process scalable in the sense that the user can work on multiple encrypted files in parallel at a time. The maximum number of files that he can open is set to INR_OPEN, which is defined in Linux as 1024. We could have instead set this value to NR_OPEN, which is defined as 1024*1024, but it does not seem practical for a single user to open so many encrypted files at once. Of course, this can be changed at any time without much hassle.

f) Maintainability

Although maintainability is not of especially great importance for what is essentially a relatively small-scale project, we have made efforts to write easy-to-understand code that is well-commented and indented in an intuitive manner.

g) Extendibility

We believe that the code can be easily extended in certain ways. For example, we can change things around to let the user specify an encryption algorithm from among any of those available in libmcrypt, let the user hide files after encryption, or let the user create a file backup on the hard disk or a cloud (although it's possible that performance may suffer if cloud file backup is used, since this implementation will be part of an instrumented library that may cause delays).

5. APPLICATIONS:

There are some potential applications for this kind project that are readily apparent. For example, a GUI text editor can be made to provide users an option to encrypt the file. Open Office or Libre Office can provide a checkbox in the same dialog box that is used to create a new file, and the status of this checkbox will be internally passed to libc via open system call as O_ECRYPT. The same idea can be applied to a command-line-based text editior, which can provide a flag to achieve the same basic capability. For example, vi editor can provide a flag "-e" as vi -e <file-name>, and thus this flag will also be internally passed to libc as O_ECRYPT via open system call.

6. REFERENCES:

[1] Stony Brook SEC Lab. "Library Interceptor." Accessed October 29, 2011. <http://seclab.cs.sunysb.edu/wsze/interceptor_guide.htm>

[2] Ask Ubuntu FAQ. "Getting Junk Data While Storing and Accessing Key." Accessed November 22, 2011. <http://askubuntu.com/questions/81208/getting-junk-data-while-storing-and-accessing-key>

[3] Stack Overflow FAQ. "When should I use mmap for file access?" Accessed November 13, 2011. <http://stackoverflow.com/questions/258091/when-should-i-use-mmap-for-file-access>

[4] Debian Bug Report Logs. "libc6: calloc returns non-zero memory areas when mlockall is being used." Accessed November 17, 2011. <http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=473812>