Tien Le

David Nguyen

CS 111 Summer 2015

Design Project – Lab 2

Introduction

Users of the ramdisk may have sensitive data that they may not want other users to be able to access. A program that may want to obfuscate what data it writes to the disk by encrypting the data itself, but we can add another layer of security by encrypting the data at the block device level.

The following is a brief description of the interface and implementation of the encryption feature for the ramdisk. For our design project, we chose to have a simple interface where a user could store data in an encrypted form, with an optional password. If the data is read without supplying this password, the user will see nothing but garbage.

Interface

At runtime, when reading or writing the user can use the **–p** argument followed by a password that is between 1 and 15 characters. A password larger than 15 characters will return an error. This option must be used with –w or –r. The password must also be provided every time the desired encrypted data is requested, as the password is not saved in the system between uses. The following code provides example inputs to the program:

```
echo test | ./osprdaccess –w –p abcde
./osprdaccess –r 5 //Will return garbage
./osprdaccess –r 5 –p abc // Will return garbage
./osprdaccess –r 5 –p abcde // Returns ‘echo’
```

This interface is completely optional and does not break the original interface, so running ./lab2-tester.pl will not cause any problems.

Implementation

When the –p argument is enabled, the module will compute the SHA-1 hash (using the linux/crypto.h API) of the password to produce a 20-byte character string. This character string is then used to encrypt the data by XORing every byte of plaintext with a byte from the hash in 20-byte blocks. This is clearly and obviously not a secure encryption scheme, but it fulfills the basic purpose of demonstrating encryption of plaintext. Afterwards, the encrypted or decrypted data will then be sent to the original block writing functions.

Overview of major changes:

osprdaccess.c

- Edited help text to display information about the –p option
- Added code in main to process the –p flag

osprd.c

- sha1_hash (line 58-82)
    - Takes an input string and computes the SHA1 hash using the linux/crypto.h module
- Encrypt/decrypt (line 86-104)
- osprd_ioctl (line 614)
    - Enabled support for the OSPRDNTERPASSWORD option, which takes a password passed into the module and stores the SHA-1 hash of that password
- osprd_encrypted_read (line 703)
    - We take a similar approach to the existing blkdev_release function, we store the function pointer of the original reads and writes in blkdev_read and blkdev_write. We then write our own read and write operations and then have osprd_blk_fops point to them by default.
    - osprd_encrypted_read is a wrapper for blkdev_read. It will call blkdev_read to obtain the data and it will then perform the decryption of the data in the module and copy the data back to the user.
    - If there is no password detected, the function will call blkdev_read, preserving original functionality as if nothing happened.
- osprd_encrypted_write
    - Similar implementation to encrypted_read, but with write operations.

Areas for future improvement

- There is a memory leak somewhere in the function that causes a kernel panic from time to time, we have not been able to isolate the issue. This error occurs only very rarely.
- The encryption is obviously awful. Understanding the crypto.h interface was difficult and most encryption algorithms required padding the data, which could be equally as difficult to implement. If we worked on this project in the future, we would change encrypt() to a more sophisticated algorithm.