Design Document: CMPS111 Asgn4

Brian De Guzman (bradeguz), Isaiah Solomon (icsolomo)

Introduction

Our assignment implements a cryptographic file system on the FreeBSD kernel at the VFS layer. Instead of encrypting the entire disk, our file system encrypts each individual file via XORing the 'sticky bit'.

Note: We edited chmod() to allow the sticky bit to be toggled without using sudo. Protectfile can be used without sudo.

Protectfile and Setkey programs

Our protectfile program takes in a 64-bit hexadecimal number as well as a file and using the hex key provided encrypts or decrypts the file. The hex key is then stored in the user credential struct (ucred) for later use by the file system. In addition, the protectfile program will XOR the sticky bit using the modified chmod() system call.

- asgn4/protectfilesFiles/protectfile.c
    - This program receives an option (-e or -d), a 32-bit hex key, and a file path. The program checks whether the file is already encrypted by checking the sticky bit using the stat() system call, calls setkey() system call to add key to user credentials if not already added, performs the AES

encryption using the code provided from class, then setting the sticky bit accordingly using chmod().

- asgn4/setkeyFiles/setkey.c
    - This program receives a 32-bit hex key as input and calls the setkey() system call to set user credential key. This is similar to the protectfile program, without the sticky bit manipulation or encryption.
- sys/kern/vfs_syscalls.c
    - int setfmode(td, cred, vp, mode);
        - This function was edited to set the userid to root when the sticky bit is set, then replaced back to current user after. This is done because editing sticky bit permissions using chmod() requires root permission.

Setkey() system call

The setkey() system call takes in two 16-bit hex keys and combines them into one 32-bit hex key. It then sets the user's ucred struct's key fields to persist after the program ends. This key is specific to the user that won't allow other users to decrypt the file without the same key.

- sys/kern/sys_setkey.c
    - This is the core file for the system call. This is where the two 16-bit hex keys are combined and where the ucred struct saves the hex key.
- sys/sys/ucred.h
    - struct ucred {

…

unsigned long k0;

unsigned long k1;

...

}

- Added to this struct are unsigned long k0 and unsigned long k1, which keep the first half of the key and second half of the key respectively.

- sys/sys/vnode.h
  - This is where we put the function declaration for the system call
- sys/kern/syscalls.master
  - This is where we added the system call with the other system calls
- sys/kern/capabilities.conf
  - This is where the system call was added to be enabled in capability mode
- sys/conf/files
  - This is where the syscall file is added to list of files

CryptoFS

This is the main bulk of the assignment. Our Cryptofs file system is a direct copy of the Nullfs, with the exception of the read and writes, which encrypts per file versus encrypting the entire disk. The Cryptofs uses the Scatter/Gather IO to transport data to be encrypted.

- sbin/mount_cryptofs/*

        - This directory was based on the mount_nullfs directory, with function names renamed from ‘null’ to ‘crypto’. This was then compiled to create an executable that mounts Cryptofs.
- sys/fs/cryptofs/*
        - This directory contains the core files for Cryptofs, copied from Nullfs. Within these files, specifically within crypto_vnops.c, the read and writes are handled.
- sys/amd64/conf/MYKERNEL
        - This file, built off of the previous assignments, adds the line ‘options CRYPTOFS’ in order to add the file system when the kernel is built.
- sys/conf/files
        - This is where the mount_cryptofs/ files as well as the cryptofs/ files are added to list of files
- sys/modules/cryptofs/*
        - This is where we built the Cryptofs module using the default Makefile, we then built the kernel to reflect the new module changes.

For the read and write calls, we looked at the other file systems, such as unionfs, to get an idea on implementing the calls. We were able to successfully implement the read and write calls in CryptoFS. We made two new functions in *crypto_vnode.c*.

- sys/fs/cryptofs/crypto_vnode.c
    - Here, we added two functions to *struct vop_vector crypto_vnodeops* :
        - .vop_read = crypto_read

        - .vop_write = crypto_write

- The read and write functions:
    - *static int crypto_read(struct vop_read_args *ap)* {}: inside this function we declared and defined variables needed for VOP_READ, then we called VOP_READ with the appropriate variables and returned the value
    - *static int crypto_write(struct vop_write_args *ap)* {}: inside this function we declared and defined variables needed for VOP_WRITE, then we called VOP_WRITE with the appropriate variables and returned the value
    - What we tried to implement:
        - The project descriptions asked to encrypt/decrypt files on the fly in the CryptoFS. Our strategy to implement this concept was to implement the read/write calls, find the file ID in order to set the sticky bit, add the AES algorithm to encrypt/decrypt the file, and then return the encrypted/decrypted file.
    - What we implemented & Problems that occurred:
        - We reached implementing the read/write calls but we ran into problems trying to find the file ID to set the sticky bit. After looking at piazza posts and researching on uio, we concluded that to find the file ID, we needed to look at the uio object. However, we were still unable to find a method to look for the file ID. Thus, we also did not implement the encryption/decryption on the fly in

CryptoFS because CryptoFS would not know which files to encrypt/decrypt.