Instructions

Adding Encryption to the MINIX file system

Overview

The main goal for this project is to use a combination of system calls and user program to implement encryption for the MINIX file system. You must implement:

- A system call that adds an encryption key for a particular user ID.
- Operating system code that applies the key to a file if *all* of the following apply:
    - The key is set for the current user.
    - The file has its encryption (sticky) bit set.
- A program that encrypts or decrypts a file and sets its encryption bit appropriately.

As with the previous projects, you'll experiment with operating system kernels, and do so in a way that may very well crash the computer (inside the virtual machine, of course—your host computer should be unaffected). You'll get experience with modifying a kernel, and may end up with an operating system that doesn't work, so you'll need to manage multiple kernels, at least one of which works.

You should also read over the general project information page before you start this project. In them, you will find information about MINIX as well as general guidelines and hints for projects in this class.

IMPORTANT NOTE: this assignment is officially due at 5PM on Friday, June 8th. However, there is an automatic 48-hour extension to 5PM on Sunday, June 10th for anyone who wants it. However, **grace days may not be used to extend the due date past June 10th at 5PM**. **Any assignments not turned in by 5PM on June 10th will receive a maximum grade of 1 point.**

Details

The goal of this assignment is to give you additional experience in modifying \minix and to gain some additional familiarity with file systems, system calls, and encryption. As with earlier programs, you won't have to write too much code, but it'll be critical to understand *where* the code goes. Most (if not all) of your kernel code will go into the mfs (MINIX file system) server and perhaps the vfs (virtual file system) server in MINIX, so that's a good place to start looking.

Encrypting and Decrypting Files

The best place to encrypt or decrypt data is at the same time as it's copied between kernel space and user space, a task that's accomplished using thesys_safecopyto() and sys_safecopyfrom() functions, which are defined in lib/libsys/sys_safecopy.c. Encryption and decryption can be done in several ways. You could copy a few bytes at a time to a buffer in the FS server and operate on them before copying them again to the user. This approach is simple, but it involves a separate buffer which you'll have to track. A second approach would involve doing the encryption or decryption in the buffer itself. Encryption is easy: just encrypt the data after you copy it into the kernel. Decryption is tougher, since you don't want to leave decrypted data in the buffer. To get around this, decrypt the part of the buffer you're copying, copy it to the user, and then re-encrypt it. Either approach (separate buffer or operate in the real buffer) is acceptable, and you maymix and match if that's easier for you.

So how do you know if a file is to be encrypted? As discussed in class, each file has *permission bits* associated with it. Fortunately (for us), one of the permission bits is rarely used for any useful purpose: the sticky bit (S_ISVTX or 01000 in octal, as defined in sys/stat.h). According to the MINIX man pages (chmod.2), this bit is only used by MINIX for directories, and you're only going to use it for files, so there should be no conflict. If this bit is set using the chmod system call (available via the command of the same name), your code should encrypt and decrypt the file automatically when it's read or written, assuming the key is available. If this bit is not set, the file is never encrypted.

Note that if no key is available for the user reading or writing the file, the file is not encrypted or decrypted either. This means that a file with the bit set can't be read if the key hasn't been set first; any calls to read or write an encrypted file without a key set should return an error (EPERM).

The encryption algorithm you'll be using is AES, which takes a 128 bit key; for this assignment, the high-order 64 bits will all be zero. You'll be using it in CTR mode, with the counter counting the offset in 16 byte chunks. Thus, to encrypt bytes 1024–1039 of the file, you'd set the CTR value to 1024/16=64. For each 16 byte chunk, encryption and decryption are done using the same function:
data ^= AESencrypt((ctr|(fileid << 64),userkey)
Obviously, integers (even long integers) aren't big enough to handle 64 bit shifts, so the above code shouldn't be used literally. Instead, you should set the low-order 64 bits (8 bytes) of the nonce (the first argument) to ctr, and the high-order 64 bits of the nonce to the file ID (the inode number). Both the first argument and second argument will be arrays of bytes, since they're both 16 bytes long.

The reason that encryption and decryption use the same function is that the result of AESencrypt() is XORed with the data. XOR it once and you get an encrypted chunk. XOR it again, and you get the original data back.

Sample code that encrypts or decrypts a file (they're the same operation!) is available as part of the AES tar file attached to this assignment.

System Calls

You'll need to write a single system call for this assignment:
setkey(unsigned int k0, unsigned int k1)
This call sets the key for the current user. The two most significant integers (half the AES key) are zero, with *k0* and *k1* occupying the other positions. Obviously, it doesn't matter *which* places are filled in the key, as long as the files aren't being shared with other MINIX systems and your setkey() system call and protectfile program (see below) are consistent about which values go where. If both *k0* and *k1* are zero in setkey(), encryption and decryption are disabled for that user. You must be able to handle keys for up to 8 users—use a static table that connects a user ID to a key.

Setting Up Encryption and Decryption

Encryption for a file is enabled by setting the sticky bit (01000 in octal). You can use the chmod system call or command (the command calls the system call) to set the sticky bit and enable encryption for a file.

Of course, merely enabling encryption doesn't encrypt the file automatically. You should write a program called protectfile that takes three arguments: the letter **e**(encrypt) or **d** (decrypt), a 64-bit key (specified as a 16 character hexadecimal number without the leading 0x), and a file name. Your program should ensure that the file is encrypted or decrypted as necessary (use the current sticky bit setting to determine if encryption / decryption is necessary) and set the sticky bit properly using the chmod() system call. The encryption and decryption should be done with the sticky bit *off* to ensure that no encryption or decryption is done automatically by the file system. Also, recall that encryption and decryption are the same function, making the process the file part of the code the same for both.

To properly encrypt or decrypt the file, you'll need the file ID (inode number), which you can obtain with the (pre-existing) stat() system call. The file itself is encrypted or decrypted using the algorithm from above that the file system uses. You're encouraged to use the sample code that encrypts a file as a base for your program.

Once you've set up the file to be encrypted, access to it should work properly if the key is set in the kernel. Of course, access to non-encrypted files should always work properly. Note that you don't need to support memory-mapped encrypted files; you just need to handle read and write properly.

Deliverables

You must turn in a compressed (gzipped) tar file of your project directory, including your design document and your report. Please ensure that the directory contains no machine-generated files when you create the tar file; you might want to use a special target (usually called clean in your makefile to handle this task). In addition, include a README file (place it in the project directory that you tar up) to explain anything unusual to the teaching assistant. **Do not submit object files, assembler files, or executables.** Every file in the submit directory that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. Also, the names and UCSC accounts of all group members must be in each file you modify.

Your design document should be called design.txt (if in plain text), or design.pdf (if in Adobe PDF) and should reside in the top-level project directory. Formats other than plain text or PDF are not acceptable; please convert other formats (Word, LaTeX, HTML) to PDF. Your design should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs. Your design document need not include details on AES beyond how to call it or use it.

Project Groups

As you already know, this project *must* be done in a group of 2–3 students; you've already been emailed the names of your group members. Only one of you should turn in the project online; the others should turn in a single file listing the names and UCSC email accounts of all project partners, and indicating which of them turned in the file.

In addition to turning in the group project, *each* project member must write a short (1–2 paragraph) summary of their contributions to the group effort. While we expect to give the same grade to every member of a group, we reserve the right to give differential grades if it's clear that group members aren't pulling their weight. This does *not* mean that everyone in the group has to contribute identically, but it *does* mean that a group member who does little or no work risks getting a lower grade.

If a group decides to use grace days to turn in their project late, **each student on the project must use a grace day for every day the project is late (e. g., a three-student project turned in three days late requires nine grace days, three from each student).** Recall that late projects (those turned in after the due date, which may have been extended by using grace days) receive a grade of 1 point, regardless of quality.

Hints

- ***START EARLY!*** You should start with your design, and check it over with the course staff.
- Experiment! You're running in an emulated system---you *can't* crash the whole computer (and if you can, let us know...).
- Look over the operating system code **before** writing your design document (not to mention your code!). Leverage existing code as much as possible, and modify as little as possible.
- For this assignment, you should write fewer (probably far fewer) than 400 lines of kernel code. Your system call should go into mfs/protect.c, and you can keep a table of keys and users local to the mfs server.
- rw_chunk() in mfs/read.c is the code that actually reads (or writes) data in a file. It's likely that placing the encryption code near the calls to the sys_safecopyxxroutines will be easiest.
- Play around with the encryption code outside the kernel first so you're comfortable with it. You should get the program to encrypt / decrypt files working at user level *before* putting encryption code into the kernel. You can either add the AES code to an existing mfs C file, or

you can add the code to the list of C files in the makefile. However, you must ensure that the AES code gets linked into the mfs server somehow.

- **IMPORTANT:** As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20~hour project in the remaining 12 hours before it's due....

Extra Credit

For 15 points of extra credit, arrange it so that setting the encryption bit (via the chmod() system call) automatically encrypts the file, and clearing the encryption bit (again, via the chmod() system call) automatically decrypts the file, both without any extra user intervention. Note that this means that, when the bit is set or cleared, the current user *must* have a key set in the kernel; otherwise, the chmod() system call should return an error when the sticky bit is set or cleared.

Doing this will break the main assignment, which does encryption and decryption manually. This means that, if you don't get this completely working, make sure you hand in the version of your code that *does* work. Obviously, don't start on the extra credit until you have the base assignment working perfectly.

Additional resources for assignment

- aes.tgz (33 KB)