6.858 Lab 5 Design Doc

Elliot Forde	Quinn Magendanz	Kevin Foley
eforde@mit.edu	qpm3@mit.edu	kmfoley@mit.edu

Non-Trivial Implementations

- Reconstructing ihandles from the VSL:

  The VSL contains only version structs for users, not for groups. Gathering every user's itable (and thus the representation of the chunk of the file system that each user created) is easy - each user's VS contains their own ihandle. But, to fetch a group's itable, we need the most up-to-date group ihandle. So, we use the ihandle from the VS that has the highest version number for each group.

- Private keys for read-only files:

  In order to ensure that only specific principals can read a file, we need to encrypt read-protected files with a symmetric key. We can't store this symmetric key in each user's client, though: if a user switches clients or if a file is group-readable, we need to be sure that the symmetric keys are fetchable from the server without the server being able to use the symmetric keys. To accomplish this, in every principal's itable, there is a mapping from principal to their symmetric key for that principal's files. For example, a user's itable has a mapping from that user to an itable-specific symmetric key, encrypted with the user's public key. For groups, the itable has a mapping from each user in the group to an itable-specific, group-shared symmetric key encrypted with each user's public key. This allows members of each group to download their share of the symmetric key from the server and decrypt it.

  We could have implemented symmetric keys at the inode level, where each inode has a shared symmetric key. But, in the event that updates to groups were made possible, every time a group member is added or removed, each group-owned inode would need to be updated, and each entry for group-owned files would need to be updated in user-owned itables, which would not be possible. In our design, when a new member is added to a group, the root user can decrypt the group's symmetric key and then re-encrypt it with the new user's public key and add it to the group itable.

- Bonus challenge: Adding rm, rm -r, and mv

  We added support for removing files and directories by adding support for the “unlink()” and “rmdir” llfuse request handlers. These handlers work similarly by first removing the specified “name” from the parent directory’s “children” list, and then removing “name”’s inode from the mapping in “tables.” This removes all pointers to the “name” file from the file system. We did not implement removal of the actual file bytes from the server because we could not find the server call which would support removals.

  We also added the ability to mv files around to different locations on the file system. We assume that the former owner and permissions for the file remain the same after the

move as before, and that a user must have write access in the location that the file is moved to.

- Bonus challenge: Encrypting Inodes

  We encrypted inodes in a similar way to encrypting the inode blocks. We added a key parameter to the “inode.load()” function and added an “inode.save()” function to abstract calls to “secfs.store.blocks” away to the inode class exclusively. Each time “inode.save” is called, it optionally accepts in a key which it then uses if the “inode.encrypted” bit is on to store an encrypted version of the inode to the server. “inode.load()” works in a similar way, retrieving the inode, and decrypting if the “inode.encrypted” bit is on.

Challenges Encountered

- Reconstructing version struct list(vsl) before file system (and .user file) has been constructed

  While reconstructing the vsl in the “pre” function at the start of a user’s session, we do not yet have access to the .user file, so the secfs.fs.usermap may not be up to date and missing some users. To solve this, we needed to get the public keys of users absent in the usermap from the key dictionary which we have in secfs.crypto. We felt that this model was a little hacky since the server is storing so much information about the users.

- Abstraction and modulation

  When we first started working on encryption, we experienced many bugs associated with unorganized encryption and decryption and improper key management. To solve this, we modularized encryption to the (block.store) and (block.load) functions and had them take in a key parameter. This allowed us to easily define where in the code encryption must take place by passing in None for the key value if we are not encrypting. Adding more modulation and abstraction to our file system would help greatly with reducing bugs and having the code base ready for change -- properties which we struggled with throughout the project.

- Perisiting UID

  We encountered a problem halfway through our project, and then again while encrypting inodes, where one test would perform an illegal operation with uid=1000 which would fail, and then the following test executed with sudo (uid=0) would report in the log file that uid=1000 was still performing the operation, so access would fail. However, if we removed the initial test, there was no issue. We were able to work around the issue on the main part of the project, but were unable to while encrypting inodes. We asked about this on Piazza on Tuesday, and the question is still unresolved.