YAEFS: Yet Another Encrypted File System

Fall 2013
6.858 Final Project

Gary Wang, Martin Rivera, Elizabeth Attaway, Alex Willisson

Abstract

YAEFS (pronounced "yeah-fs") is a networked encrypted file system written in Python, allowing users to securely store and share files on an untrusted server. YAEFS has users, groups, and permissions, similar to traditional UNIX permissions. File system information such as unencrypted file contents, file names, and directory names are all only ever handled client side, the server never needs to have access to any of these elements. The server has access to the file system structure, but it stores random node IDs instead of file and directory names. The server also has complete access to user and group permission fields for every file and directory, along with the permission bits (read and write permissions for users and groups). The YAEFS client we developed uses the FUSE library, and once acts almost exactly like any other part of the file system. The main differences which users will see during normal use is that there is no chown command, and there is a special command for handling groups.

Network Encryption

The bodies of the HTTP requests and responses are encrypted using a schema that declares three encryption levels:

- No encryption: provides no security whatsoever. It is only used by the server to transmit its public key.
- Server/user public key encryption: provides the security of RSA. It is only used by the register and login commands since RSA's performance is subpar to the performance of symmetric encryption. A successful login attempts results in the exchange of a symmetric key called the session key.
- Session key: The session key is used for all the communications between the user logs in and logs out.

Since the whole body of the request is encrypted, the parties do not know which protocol and key were used to encrypt the request. To go around this limitation we add special HTTP headers EFS-Encrypted and EFS-KeyId that tell whether the request is encrypted and transmit a unique id of the key used to encrypt the message. The protocol used can also be inferred from the key id.

Server structure

The server is designed to increase security when it is behaving correctly, and to cause minimal damage if it is compromised. The server is entirely agnostic regarding the contents of files, it never needs to read any of the contents of a file. This makes it easy to handle encrypted files, since the server does not need any information from the file itself.

Once a Request from a client has been received, the Request is passed to the correct function via server.py. All keys are stored in mongodb and retrieved through the key service defined in keys.py. authorization.py checks to make sure users have authorization for commands which they are calling, before the actual command is executed. All other commands (new files and directories, moving and copying files or directories, permissions, reading and writing files, etc.) are passed along to fs.py. fs.py uses another set of mongoDB tables to store file and directory IDs (called node IDs), which files are in which directories, file owners, and groups for files. fs.py handles all the logic for ensuring that a user has the proper permissions for the node ID requested, adding an extra layer of protection on top of the encryption of the files. Permissions in fs.py are designed to mimick UNIX permissions, but are not identical. Permissions are inputted as a three byte string that appears to be in the same format as UNIX file permissions, taking advantage of a familiar format. However, the "other" byte and execute permissions are ignored. Allowing users who both aren't the user and aren't in the group owning a file access to that file would defeat the purpose of encrypting the file, as an unencrypted version of the file would have to be stored.

Files and directories are both stored on the server as nodes and are essentially indistinguishable through looking at the database. Every node has a parent node, listed by ID. Directories are nodes which have a list of other nodes which they are containing. Files are nodes with a file containing the encrypted data. When a user wishes to create a new file, it sends a request to the server to allocate a new node, specifying the permissions and parent node of the new node. The server checks to make sure the client has permission, and either returns the node ID (a random 16 byte number represented by a hexadecimal string) of the new node, or an error. Whenever the server receives a write request for a file, it creates (or updates) a file whose name is the same as the node ID in the same directory as the server was run in. Because the file was encrypted client-side by a key the server does not have and the node ID random, the server has no information regarding the file's name or contents.

Client structure

A user's primary way of using YAEFS is through calling mount.py with the configuration file (containing connection data), logging in, and then using the file system normally. Once mount.py is called, users can access the data in MOUNTPOINT/username, where MOUNTPOINT is defined in the client's settings file and username is the username used to log in. If it is the first time the user has connected to a server, the user will be prompted to verify the server's public key, much like SSH when connecting to a server for the first time. There are several new commands YAEFS uses for group management, prompted through difficulties related to FUSE and only group IDs being passed through by syscalls, instead of group names. To alleviate this issue, YAEFS provides a utility called user_management.py which allows users to submit commands to the server. In order to work, the client writes the session information to a file called .efs_session in the user's home directory and uses those value to communicate with the server bypassing FUSE. The commands supported by user_managment.py are:

- chpasswd: changes the password of the user
- mkgroup: creates a group with owner as the client running the script
- rmgroup: removes a group
- adduser: adds a group to an user
- rmuser: removes a user from a group

Our YAEFS client encrypts files with a randomly generated symmetric key before sending them to the server. The symmetric key itself is encrypted twice, once with the public key of the owner, and once with the public key of the group of the file. The two versions of the encrypted symmetric key are stored with the encrypted contents of the file before being sent to the server.

A disadvantage to this design is that changing the group of a file requires the file's owner to download the whole file, decrypt the owner key with his private key, encrypt the decrypted key with the public key of the new group, and replace the old group key. The fix for this problem is simple: have the server store the file keys separately from the file. When the client issues a change group command, the client would only need to update the stored keys, because every file has it's own unique symmetric key.

As the server does not know the names of files or directories, the client needs to store information linking the node IDs which the server understands, with file and directory names which a user can understand. To do this, we have a structure file for every user. The structure file contains file and directory names linked to node IDs, along with the directory tree. This file is also how we verify that the server is not modifying files, rolling back files, or substituting one correct file in place of another (i.e. if the user had files A and B, the user requests A, and the server may try to serve the user file B). It stores file hashes, which are checked with files are read from the server. The structure file also contains a version number, which is updated and stored on the client side to make sure the server never rolls back the structure file itself. The structure file is encrypted so that only the user can decrypt it. On top of the structure file unique to each user, group also has a structure file which all users in the group have a copy of. This group structure file lets a user know what node IDs exist that members of the group have access to, along with the file and directory names. Groups have their own public/private key pair which are distributed to all members of the group. User structure files are encrypted with the public key of the user, and the group structure files are encrypted with the group’s public key. When a user is added to a group, the user sending the request to add the other user will encrypt the group's private key with the target user's public key. This way, the user newly added to the group will have access to files the group has access to, without the server ever having a chance to see the group's private key.

Groups do provide a potential security problem when a user is removed from a group. Our design keeps using the same key pair for a group as users are added and removed from it. This means that a malicious user could retain a group's private key after being "removed" from the group. Fortunately in the case where the server is not malicious, the malicious user will not be able to retrieve more files owned by the group, so having the group's private key is significantly less useful. In the case that the server is malicious and discovered a group's private key, all files on the server encrypted with that key should be considered compromised. Attempting to re-encrypt files is a useless exercise, as the server could simply retain old versions of the files encrypted with keys the server has access to. This does mean that any future files owned by the compromised group would still be compromised, but if the server and a group's private key was compromised, the server should not be continue to be used. A workaround to a group's key being compromised is to create a new group and copy files over.