6.858 Final Project: The Daniel, Nathan, and Daniel File System (dndfs)

Group: Daniel Gray (haiiro), Nathan Arce (codetaku), Daniel Ziegler (dmz)

Problem

Storage and synchronization of files between machines and users is a common desire, but comes with significant security risks. Traditional networked file systems rely on the server to provide access control on files, but if the server is compromised, the attacker can access and modify all of the files. Providing security guarantees in the face of an untrusted server is the next step in the evolution of file systems.

Assumptions / not solving these problems

- The user's machine is not compromised and the user's private key is not known by adversaries
- Securing a connection between a client and a server with TLS prevents all network-based attacks
- Users who want to share files using this system exchange public keys through some external mechanism beforehand

Threat model

We have two threat models to consider. In either case, an adversarial server can inspect any data sent to it and modify that data in any way that it wishes. The difference is how the server deals with malicious clients:

1. In the first model, we have a server enforces our protocol, making sure users do not write to blocks when they do not have write permissions
2. The the second model, the server allows allows malicious users to write to any block, whether or not they have write permissions

Guarantees

Our design provides the following security guarantees, given our first threat model.

A server cannot:

- Read any file
- Determine the filename associated with any file
- Determine the size of any file or directory*
- Determine the directory structure*
- Modify a file or directory without detection

- Substitute one file for another without detection

* Starred items may be inferred in part from access patterns

A user cannot:

- Read a normal file for which they do not have read permissions
- Write to a normal file for which they do not have write permissions
- Read the entries in a directory for which they do not have read permissions
- Add entries to a directory for which they do not have write permissions
- Read a file in a directory for which they do not have execute (traversal) permissions
- Delete a file or directory that they do not own

In our second threat model, our design still provides most of the above guarantees. However, a user can now modify or delete any file in a directory for which they have traversal privileges, but this modification will be detected. Slightly worse, they can take ownership of any file that they already have permission to read, and given ownership, can grant themselves write permissions for that file. However, they cannot assign ownership to any user whose private key they do not know. We argue that this is a detectable change: the real owner of the file can notice that they no longer own the file.

Design

Overall structure

Our design has server-side and client-side userspace programs that communicate over a TLS-secured network socket. We use fully client-side cryptography, with server signature verification as an additional measure, to hide secret information from both the server and malicious clients and still authenticate writes to the server. Furthermore, files and directories are abstracted on the server into series of “blocks” stored as key-value pairs on the server. After separation of a file into some number of blocks based on the file’s total length, each block is padded to a precise length and wholly AES-encrypted, then signed with a dynamically-generated ECC key. This mechanism is what allows us to hide even file length from the server: because the public keys on each block even within a file appear different, all blocks merely appear to be 48-kilobyte chunks of nonsense to a server that does not have the read key for the relevant file. The directory structure and filenames are dissociated from blocks on the server, as the server is given an “address” for each block that is a hash of the filename, the traversal key of the file’s containing directory, and the block number within the file.

Detailed explanation of cryptography

We used elliptic curve cryptography for all of our public-key needs, and AES for all symmetric-key encryption. ECC keywraps are fairly small, certainly smaller than RSA, so we were able to easily confer permissions by encrypting a key for each permission each user has on a folder with that user’s public key. Each user only has a public/private keypair attached to their name/userid directly, and all other keys are encrypted on the server.

Privileges on non-directory files are separated into two categories: reading and writing. The write key on a file is an ECC private key (string) used for authenticating writes to the server. However, the actual private key on a given block is dynamically generated based on the file write key and the block number within that file that the block corresponds to. In this way, blocks of the same file look uncorrelated on the server because they do not have the same attached world-readable write key. The read key on a file is an AES key. It's both used to CTR-mode encrypt a file's contents and stringified to ECC-sign a block's encrypted contents. As such, there are actually two signatures attached to a file: one is from the write key that the server verifies itself, and the other is from the read key, which can let a user know if the file it is reading was last written by "gibberish", rather than just decrypting and accepting gibberish. This system is in place so that even if the server is malicious, it cannot trick users into accepting its data. Furthermore, while someone that has write but not read privileges on a file can theoretically corrupt data on the server due to their write key access, they cannot write "sensible" data to a block, which we feel makes sense, considering they have no idea what was in that block in the first place.

Privileges on a directory are more complicated. While there are three categories (reading, writing, and traversing/executing), there are actually five keys. One key is to read blocks, equivalent to the AES read key from Files. This is given to anyone with read or write privileges. Another is to sign blocks, equivalent to the ECC key from Files. This is only given to those with write privileges. Another is to write directory entries. This is an ECC public key that is used to ECC-encrypt individual directory entries into the content field of a block. Of course, this is given only to those with write privileges. The corresponding private key is given only to those with read privileges. Of course, this means that one with read privileges can generate the public key and forge directory entries that look correct to other people with read privileges, however, they **cannot** authenticate those blocks to the server, which means that this system is no less secure than our system for Files: both the server and a user with read privileges have to collaborate in order to corrupt data in a way that can fool other users but is not supposed to be allowed. The fifth and final key is the traversal key, for execute permissions. This serves as both an AES key for reading the inode content of files and subdirectories, and a salt that allows you to correctly hash the addresses of files and subdirectories in order to figure out where to ask the server to poll from (this is different from read permissions on a directory, which merely allow you to view file and directory names). In this way, we've tried to best emulate the UNIX filesystem model, but with cryptography instead of kernel kludges, and in a way that gives the server virtually no information about what is being stored on it.

Future work

It should be possible to extend the client-side metadata to support something more like standard Unix user-group-other permissions, instead of the current ACL-style permissions.

Known problems with current implementation

For simplicity, we currently use Python's insecure "pickle" package to serialize and deserialize objects. An implementation for real-world use will want a safe custom serializer/deserializer limited to specific objects.

Our server cannot handle parallel connections, making it susceptible to simple denial of service attacks and slowing it down when dealing with multiple or clever clients. A real-world implementation would want a multithreaded server with careful synchronization to handle multiple concurrent requests from one or more clients. In this case, we would need to more carefully think about how to handle intra-file freshness guarantees, due to the server's abstraction level, which involves data blocks for the same file being uncorrelatable, sans access patterns.

A production server should probably have a proper CA-signed TLS certificate. A production client should probably not be hard-coded to connect to localhost:11111.