Recitation 3 Notes:
Using hashing and encryption to secure a file system
March 1, 2013

- File system security has been a productive research topic in the file systems community: CFS (crypto FS), SFS (secure FS), SUNDR, NCryptfs, and others. We can already work through some of the basic techniques in these systems using what we learned in class, hashing and encryption.
- System setup: a client and a server. The client is trusted and wants to use the server to store its private data. The server is not trusted. The client uses a get/put interface to fetch and post items from/to the server.
- What can an untrusted server do? Reveal client data to other parties (confidentiality threat), modify data (integrity threat), discard updates to data and instead keep old data (integrity threat, also called freshness).
- What security properties do we want: confidentiality, integrity.
  - Confidentiality: no unauthorized reading of data $\Leftarrow$ enforced by the property "server does not see plaintext data"
  - Integrity: no unauthorized change of data
- 1. Confidentiality
  - How to provide confidentiality? Encryption.
  - For practical reasons (to avoid storing large one-time pads), we prefer a reusable encryption schemes so OTP does not work here.
- 2. Integrity
  - How to protect integrity with what we learned in class so far? Merkle trees.
  - Now we worked out the algorithms for get and put that include Merkle hashing and verification.
  - Why can't the server modify a file? What property of the hash prevents against? Collision-resistance.
- 3. Integrity of history of updates: svn, git via **chained hashing**
  - Consider only one file for simplicity.
  - User may want to ask for version j of the current file, but server may provide an incorrect version.

– The client could store a hash for every version of the file and then compare the hash of the file downloaded from the server with the stored hash. This strategy is inefficient because there could be many versions.
– Use chained hashes. Let F_i be the file at version i, H_i its hash and ch_i its chain hash. The chain hashes are computed as follows:

$$F_0, H_0 = \mathsf{hash}(F_0), \mathsf{ch}_0 = H_0$$
$$F_1, H_1 = \mathsf{hash}(F_1), \mathsf{ch}_1 = \mathsf{hash}(H_1, \mathsf{ch}_0)$$
$$F_2, H_2 = \mathsf{hash}(F_2), \mathsf{ch}_2 \ldots$$

Generally: $\mathsf{ch}_i \leftarrow \mathsf{hash}(H_i, \mathsf{ch}_{i-1})$

– Note that ch_i depends on entire history: $\mathsf{ch}_{i-1}, \ldots, \mathsf{ch}_0$, so $F_{i-1}, \ldots, F_0$. By collision resistance of the hash, the server cannot come up with a different history of a file that matches the current chained hash of the file.
– We then worked out the detailed protocols for getting an old version and for updating a file (put), and the verifications the client must perform to check correctness.