CS 161 - Project 2: Design Document

Brent Sienko
Itai Smith

Section 1: System Design

Guiding Questions

How is each client initialized?
We first generate a deterministic UUID as the Data Store key for our new User Struct as well as deterministic Encryption/Decryption and Verification keys so that a user can retrieve his User Struc. Because it is given that the input *password* has sufficient entropy, we only needed to add some deterministic salt to create three secure keys (e.g. *username* + *password* + "deterministic salt"). We then create two pairs of RSA keys, one pair for Digital Signatures and the other for Symmetric encryption/decryption, and store them in our User Struct and store the public keys in the Key Store, used for user file sharing. To manage the various files a User has access to, we create two dictionaries: FileTable, which maps filename to file UUID, and KeyTable, which maps filename to the keys necessary to decrypt and verify files retrieved from the Data Store. After creating and filling the User Struct, we json.Marshal(), encrypt, sign, and finally store the User Struct in the Data Store.

How is a file stored on the server?
Every time StoreFile(filename, data) is called, we create a random UUID and a pair of random symmetric keys (encryption and verification) for that file. We store the random UUID in the user structure map: "filename > UUID", and the keys in the user structure map: "filename > symmetric keys". That is, if StoreFile is called again with the same filename, the data sent in will be stored at a different UUID with new random symmetric keys. The old data associated with this filename will be lost. We then use authenticated encryption with the previously mentioned keys to encrypt the file and store it at the random UUID generated for it.

How are file names on the server determined?
Filenames are held inside the encrypted user structure in two maps. One maps plain filenames to the UUIDs in which they are located on the server. The other, maps plain filenames to the symmetric keys used for the authenticated encryption / decryption for that file.

What is the process of sharing a file with a user?
For every fresh file stored we create an "updates structure" that holds a list of the UUIDs of all updates made to this file (by owner and other users). It does not include any information about the original file. When we share a file with another user, we create a "sharing structure" that holds the UUID of the shared file and keys required to decrypt and authenticate it. This structure also contains the respective information for the "updates structure" of the shared file, so the new user has the ability to append. We send the UUID and keys of the sharing structure (which are also generated randomly) in the magic string, which we use RSA encryption and signing to send over the insecure channel. The new user extracts the UUID and keys of the sharing node from the magic string, and saves them as the UUID and keys of shared file in their user structure file and key maps. When they will want to load the shared file, we recurse using the information in the sharing record and retrieve the data from the original file (appended with all updates made to it).

What is the process of revoking a user's access to a file?
Since filenames are not saved on the server, and users who are not original owners do not have the UUID of the original file saved in their map of "filename > UUID", only the original owner of the file can revoke access to it. We delete from the server the load the file with all updates made to it. Delete the original file and its updates structure from the server, and repeat the process described in StoreFile.

Testing Methodology

Along with testing many file-share functionalities, we focused on writing tests that ensured appropriate system response (i.e. fail-safe defaults, complete mediation) to different attacks such as user struct and file corruption. Our test suites check for the attacks mentioned below and more.

Section 2: Security Analysis for Potential Threats

1. File modification: Every file is encrypted and authenticated using a unique pair of keys that are generated randomly. The only way to gain any information about a file, including the filename itself, is through the user structure, which can only be accessed if one possesses the correct combination of username and password. Any attempt to modify a file on the datastore will result in failure in authentication and trigger an error. Any attempt to perform brute force search for filenames is futile since they are not present on the Data Store in plain or hashed form.

2. Swapping files: If Eve tries to swap to files on the Data Store, and leave a malicious payload in a user's UUID, we will trigger an error. Every user will try to decrypt and authenticate each file using symmetric keys that are unique to this file and generated randomly. In the case where a file's HMAC doesn't match the expected, we will not continue with loading the file and trigger an error.

3. User Struct corruption: Because each user struct is Encrypted and Signed with Keys deterministically created based on the user's username and password, each user is stored in the Data Store with Integrity and Authenticity. If an attacker somehow gained access to the Data Store and and tampered with a User Struct, upon calling GetUser() and attempting to decrypt and check that the HMAC of said User Struct matched the tag, our system would throw an error because both Integrity and Authenticity of the User Struct were compromised.

4. Man in the middle attack: Consider the situation where Alice is attempting to share a file with Bob and an attacker,, Eve, interferes. Suppose she somehow obtains the *magic_string* in attempt of gaining access to this file. Our scheme, which uses RSA Decryption and Authentication keys, would not allow Eve to decrypt unless she had Bob's private RSA key, and would therefore not leak any information.