CS 161 Computer Security

2020-10-17

Project 2 Design Doc

A Secure File-sharing System

on a Hostile Storage

1. System Design

a. How is a file stored on the server?

Files are stored by users. To first store user info, we encode a User struct into JSON, symmetrically encrypt it with k_user_encrypt, pad it until it is divisible by 16 (AESBlockSize) according to PKCS #7, HMAC the ciphertext with k_user_auth, and store the HMAC appended by the ciphertext on Datastore, at the storage key ID_user. To generate those keys, we first deterministically derive k_password with PKDF from the password, salted by the unique username s.t. different users have different keys even if they share the same passwords. We then deterministically derive the other keys with HKDF from k_password, with the purpose array set as the encoded version of string 'user_encrypt' for k_user_encrypt, that of 'user_auth' for k_user_auth, and 'user_storage' for k_user_storage. ID_user is the UUID of the HMAC of the (bytes of) username with key k_user_storage.

We also generate a PKE keypair, k_pub and K_private, and a DS keypair, k_DS_pub and K_DS_private, for each user for sharing. We publish k_pub on Keystore at the storage key k_pubkey, which is an UUID generated from the first 16B of the hash of the unique username appended by string ‘public_key’. We publish k_DS_pub at the storage key k_DSkey, which is another UUID generated from the first 16B of the hash of the unique username appended by string ‘DS_key’. We store K_private and K_DS_private in the secured and authenticated User struct.

For a user to retrieve their User struct, they enter their username and password to generate ID_user through the aforementioned deterministic process. If ID_user exists and the password is correct, the user_struct will be decrypted and depadded (according to PKCS #7) upon retrieval.

To store a file, a user calls User.StoreFile() with their User struct. The file is encoded into a byte array and splitted, each volume encrypted and then its ciphertext authenticated by its MAC. The split is in blocks of 1GB to ease future appending. The last block is padded to 1GB with the length of the pad in bytes (following PKCS #7). The encryption is through AES-CBC encryption, with the 16B IV generated by a RNG for every volume and the key k_volume deterministically derived through HKDF from k_file, with the purpose array set as the encoded version of string ‘volume_encryption’ appended by the index of the volume. Therefore, we have different IVs and keys for all volumes. k_file is a random 128b (16B) symmetric key from our RNG. The HMAC key is k_volume_MAC, again deterministically derived through HKDF from k_file, with the purpose array set as the encoded version of string ‘volume_authentication’ appended by the index of the volume. We define a Volume struct that contains the ciphertext and the HMAC as two arrays, and the number of padded bytes as uint32.

We fetch k_pub and k_DS_pub for the user, append k_file to 16B of random padding in front to prevent IND-CPA, PKE encrypt the 32B k_file_front_padded with k_pub to output pke_k_file, and Digitally Sign pke_k_file with K_DS_private to output ds_k_file. We store ds_k_file and pke_k_file in a SignedKey struct, and store it in a marshalled map of SignedKeys on Datastore at ID_k, where ID_k is a random UUID. The key in the map is the username. We store ID_k in map AES_key_storage_keys in the User struct, with both keys being the filename, and update the User struct on Datastore.

The encrypted volumes are then put together as an array of Volume structs, marshalled again and stored on Datastore. The storage key is ID_file, which is generated from the hash of the string representation of ID_k converted to a byte array.

If a user calls StoreFile on a filename that already exists, the file content is overwritten.

b. How does a file get shared with another user?

The sharer uses the ShareFile method, which will generate an access token to be sent to the sharee and is called whenever the owner of the file selects a file to be shared. The token only permits the sharee to read, write and share the file, and nobody else.

We first retrieve the master key, k_file. We find the storage key ID_k of all encrypted AES keys (k_file's) for the file in a map in the owner's User struct, find the owner's SignedKey with their username, and verify and decrypt it with the owner's public DS key and private PKE key.

We then PKE encrypt k_file with the recipient's public key k_pub generated when the user was initiated, Digitally Sign the ciphertext with the sharer's k_DS_private, create a SignedKey struct with the signature and the ciphertext, and add it to the SignedKey map at ID_k, with the recipient's username as the key. We then marshal and store the updated SignedKey array at ID_k. The token is a SignedKey struct containing two variables: ID_k encrypted by the recipient's k_pub, and another version that is Digitally Signed with the sharer's k_DS_private. Lastly, if the sharer owns the file, we note the direct recipient of that file in a map in the owner's User struct.

When the recipient receives the token using ReceiveFile, they will first come up with their own filename for it. If they already have a file with that name, err. If not, they verify the signature with the sharer's k_DS_pub and decrypt the message using their own private key K_private. If verification works and the file exists, a mapping from an arbitrary new filename for the recipient to ID_k is added to their User struct. If verification fails or the file does not exist, return an error.

c. What is the process of revoking a user's access to a file?

The owner will select the user whose access will be removed and then call RevokeFile. This method removes the ability to read, write and share for the selected user.

It first checks whether the user owns the file. If they do, it then generates a new 16B symmetric key k_file, and gets the segmented plaintext like in LoadFile. encrypts and authenticates the plaintext with it like in the storing process, encrypts k_file like in regular sharing with every direct sharee's public keys k_pub except for the revokee's (according to a map in the owner's secure User struct), signs all those ciphertexts, and stores the updated SignedKey array back at ID_k.

Returns an error if the selected user does not own the file or if file does not exist.

d. How does your design support efficient file append?

Since the file will be segmented into volumes of 1GB in size, we won't have to decrypt or authenticate the entire file. Instead, we only need to decrypt the last 1GB of data, append to that data, then re-encrypt it. This will be faster than if we had to decrypt the entire file.

2. Security Analysis

a. Attack on Storage

If user A stores a file, user M might store their own file to have the same name as the original file's. Our design prevents problems by not incorporating filenames in storage locations etc, and only stores it in a secure User struct, which is part of the Trusted Computing Base.

b. Attack on Sharing

The attacker has the ability to read the token in transit. Thus, they can use it to gain unauthorized access to shared files by . In order to prevent this, our design uses the public key of the recipient (stored in Datastore) to encrypt it so it can only be decrypted by the user to which the token is intentionally sent.

c. Attack on Revocation

User A could share a file with user B then revoke access. User B could then try a denial of service attack by arbitrarily changing contents of the file. To detect that, we have designed HMACs for each volume of the file, where the keys are derived from k_file, a secure master key whose possession is equivalent to legal access. It is an unpredictable random number and is stored elsewhere on Datastore, encrypted by RSA. Therefore, M would not create valid MACs for the modified file, and A will detect any change after an unauthorised modification and MAC authentication fails.