Jonathan Briggs and Vedaank Tiwari
Computer Security - Fall 2018

Project Two: Secure File Store – Design Document

Overview:

1. Structs
    a. User
        i. User Struct
            1. Username: contains the HMAC of the username with the password as the key
            2. Password: Argon2Key of password with salt of username to prevent dictionary attacks.
            3. Salt: Username string
            4. PubKey & PrivKey: RSA public and private keys, respectively
            5. Shared & Created: maps of files shared with or created by user, respectively
        ii. Pair Struct
            1. Holds encrypted data in the Data field and the corresponding MAC of the encrypted data in the MAC field.
    b. Files
        i. File Struct
            1. Stores a file's UUID, its unique encrypted key, its MAC key, and its creator's username.
        ii. FileNode Struct
            1. Contains the total length of the file in bytes, number of edits made to the file, and the individual sizes of each edit. This enables the file to be loaded properly after edits have been made.
        iii. FilePair Struct
            1. Holds encrypted data in the Data field and the corresponding MAC of the encrypted data in the MAC field.
    c. Sharing
        i. sharingRecord Struct
            1. Stores a FileNode (that has been converted to a byte array) for convenient, streamlined, low-cost, and ultimately more secure sharing.
        ii. Packet Struct
            1. Contains a pointer to a sharingRecord and the RSA signature for the packet.
2. Key Functions
    a. InitUser
        i. This function populates the User struct with the relevant information. The Username field is set to the HMAC of the passed in username and uses the password as the key. The Password is stored using Argon2 and a salt of the username. Then, we generate the RSA keys for the user, initialize the maps for created and shared files, and populate the userdata. If this proceeds without error, we set the keys for the user in the keystore and call the SendToDataStore() helper, which nicely packages up the userdata by converting

it into bytes (using Marshal), encrypting it, storing it in a Pair struct, and storing it in the Datastore at the proper path.

b. GetUser
    i. Given a username and password string, this function queries the datastore for the relevant userdata by reassembling the information that would be stored for the passed in variables like InitUser and fetching from the Datastore. After confirming that the data has not been tampered with, it is decrypted and returned.
c. StoreFile
    i. This function populates a new File struct and stores it in datastore securely. It assigns each file with a unique identifier and generates random bytes for the encryption key and mac key. We also create a FileNode and call a helper function to populate it. Then, the file is encrypted, an HMAC is generated, and these two values are stored in the datastore.
d. AppendFile
    i. After checking permissions, we proceed to reverse engineer the relevant calls to the Datastore that we created in StoreFile based on the passed in filename. Next, we fetch the encrypted file from the datastore. If it exists and has not been tampered with, we modify the file's associated FileNode to reflect the updates, encrypt the new data, and append it to the fetched efile before storing the whole thing again in the same location.
e. LoadFile
    i. After checking permissions, we fetch the file from the datastore and obtain the information in the associated FileNode. If we deem that the file has not been tampered with, we proceed to unpack all of the data. Originally, we looped through a bunch of "next" pointers, but then the File struct became too large to marshal and encrypt with RSA. So, instead, a helper function is called that takes advantage of a more streamlined File and FileNode struct which tracks the number of edits and their sizes. This way, we can loop through all the individual appends, verifying each one and putting all the decrypted data back together to be returned.
f. ShareFile
    i. After checking permissions, we fetch the file from the Datastore. Then, the file is marshaled and the user's RSA public key is retrieved from the Keystore. Next, we encrypt the file using the RSA key, sign the efile with their private key for confidentiality and integrity, and assemble a Packet struct. This is marshaled and converted into a string before being returned as the "msgid."
g. ReceiveFile
    i. This function unmarshals the received Packet, fetches the receiving user's RSA keys to verify the message, and decrypts the sent packet. Then, the file is added to the map of files that is accessible to the user.
h. RevokeFile
    i. After checking to confirm that the user calling RevokeFile is the owner, we load the original file data. Then, we re-encrypt it with a throwaway key before saving

it back into the datastore. We can save the MAC, though, which is useful for using previously defined helper functions. Then, we simply outsource the file to StoreData, which takes care of the rest by storing the file in the Datastore as if it were brand new.

Testing:

To test the functionality of our user initialization and loading, we simply need to confirm that storing for a given username/password combination will return the same user on a call to LoadUser(). For the basic file operations and methods, we need to check that no errors are produced when trying to store, append, or load. Further, we need to be positive that our outputs remain correct after a file has been edited by any users with access to it. We can also check to make sure that our system identifies and detects tampered or modified file data. Next, we confirm that our message is send to the datastore properly, received without issue, and more importantly, that it is loaded safely and correctly by the end-user. Lastly, we checked the robustness of our implementation of file privileges (RevokeFile). To test this, we simply need to confirm that only a file's original creator can perform this function and that it works as intended (no other users have access after the function executes).

Security Analysis:

1. Password Dictionary Attack
    a. The username and password are not stored directly in the User struct. Rather, the username field contains the HMAC of the username with the password as the key and the password field stores the Argon2Key of password (with salt of username). This prevents any entity with malicious intentions from mounting a dictionary attack using common passwords or any precomputed hashes of these.
2. MITM Attack
    a. Any attempt to eavesdrop when a file is being shared will fail due to the utilization of the RSA protocol in the ShareFile and ReceiveFile functions. Any attempts to change the contents of the file will be thwarted by the RSA signature since the attacker has no way of obtaining the private key of the file sharer. This is because the User struct is encrypted in the Datastore. Further, the attacker has no way of decrypting the file without accessing the private key of the user who is being sent the file for the same reasons.
3. Filename Dictionary Attack
    a. Similar to the password dictionary attack, the attacker cannot mount an attack by using known filenames or their hashes. To avoid this, we generate a unique identifier for each file and encrypt all data needed to access the file before putting it in the Datastore. This prevents an adversary from obtaining any information about any files in the Datastore.
4. Data Swap Attack
    a. The ability to guard against this was not actually intended and arose as a result of a change to the way that we modified our File/FileNode struct in order to fix a bug in ShareFile's RSA encryption methods. This also required a change to AppendFile, which made us realize our luck. Since we are storing data and

structures for file retrieval and loading in separate locations in our Datastore, an adversary may deduce the relationship between these two locations and try to swap two files and their entries. However, we are MACing all of our files to ensure integrity. Therefore, any data swap attempt will trigger an exception and be detected.