Vaibhav Bhat
Sai Mandava

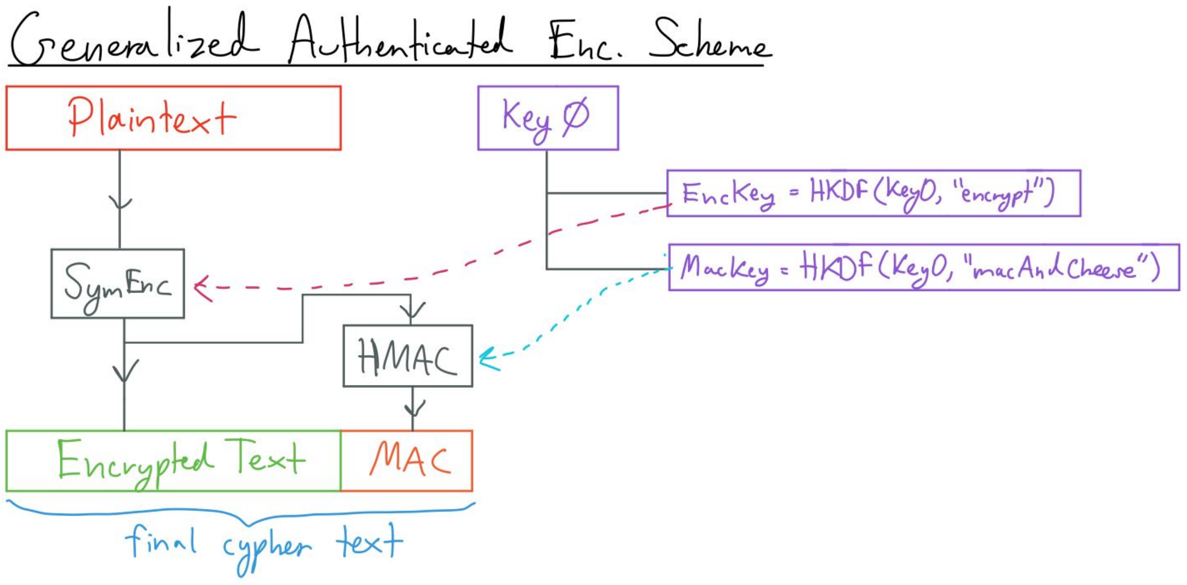
CS161 Proj. 2 Design Doc *(detailed version)*

Section 1: System Design

1. Generalized Authenticated Encryption Scheme: (GAES)

   In general, takes in Plaintext and a Key and encrypts as detailed in below diagram. This scheme guarantees confidentiality and integrity of the data as well as authentication using MACs. **ALL uploads/downloads to/from Datastore will use the GAES functions.**

   - Encryption/upload be implemented as a helper function that takes in (UUID, plaintext, key) and uploads the final (UUID, ciphertext || HMAC(ciphertext)). Keys are different for both and derived from the passed-in key.
     GAES Upload guarantees confidentiality by using symmetric encryption with a secret key.
   - Decryption/fetch will also be a helper function that takes in (UUID, key) and returns the plaintext. Errors if tampering detected.
     GAES Download checks for integrity by checking the MAC.


Generalized Authenticated Enc. Scheme
Plaintext
Key Ø
EncKey = HKDF(Key0, "encrypt")
MacKey = HKDF(Key0, "macAndCheese")
SymEnc
HMAC
Encrypted Text
MAC
final cypher text

2. Structs:

1. **User Struct:**
   - a. **Username**
   - b. **K0:** `// K0 = A2K(pswd, username, 32 bytes)[:16]`
     We store this because they'll be scenarios where we want to update the user struct but don't have access to username and password. This is equivalent to storing Username and Password in the struct and recalculating K0 every time.
   - c. **UUID:**
     `// tempKey = A2K(pswd, username, 32 bytes)[16:32]`
     `// UUID = UUID(HMAC(tempKey, username)[:16])`
     The user's calculated UUID at which this struct is stored in Datastore
   - d. **MyFileTable:** A map from filename ➔ [fileHead UUID, fileHeadMasterKey]
     In reality the limitations of json.Marshal forced us to make two separate tables, but the abstraction here serves us better.
   - e. **SignaturePrivKey:** Private key for Digital Signatures *// For sharing*
   - f. **AsymmPrivKey:** Private key for Asymmetric Encryption *// For sharing*
2. **FileHead Struct:**
   The 'head' of a file. Points to all the pieces of the file so the file is augmentable. Will make revoking share access easier; to share, we will just share the FileHead UUID concatenated with the `FileHeadMasterKey` used to derive the keys to encrypt the FileHead and encrypt share signatures related to that file. To revoke access for all users, we will just change the UUID and master key of the FileHead rather than changing the UUID/key of all segments.
   - a. **NumSegments:** Integer number of segments in this file
   - b. **SegmentTable:** Map from segment # ➔ [fileSegment UUID, segment key]
     In reality the limitations of json.Marshal forced us to make two separate tables, but the abstraction here serves us better.
3. **FileSegment "Struct":**
   Not actually a struct, just the concept that each segment of a file will be stored separately in Datastore as (random UUID, byte content of segment).

3. Functions:

1. **InitUser(username string, password string) (userdataptr *User, err error):**
   We calculate the fields of the User Struct as detailed above, and store the public keys (asymmetric key and signing key) in the Keystore at `username + "/asym"`

and `username + “/sign”`, respectively. We upload the new user struct to Datastore using GAES. So, here’s the full process:

   a. Calculate `K0` and `UUID` by slicing and processing the 32-byte A2K output, and put them and `Username` in the struct.
   b. Generate an empty map in struct to be the `MyFileTable`.
   c. Generate PKE keypair. Put private key in struct as `AsymmPrivKey` and public key in Keystore with key `(username + “/asym”)`.
   d. Generate Signature keypair. Put private key in struct as `SignaturePrivKey` and public key in Keystore with key `(username + “/sign”)`.
   e. Upload User struct to Datastore using GAES, with UUID = `UUID` and key = `K0`.

2. **GetUser(username string, password string):**
   We recalculate `K0` and `UUID`, and then retrieve the user struct from Datastore using GAES, which checks for integrity/authentication as we’ve established.
3. **StoreFile(filename string, data []byte):**
   Since filenames have low entropy, we create a randomly generated UUID `FileHeadUUID` and a randomly generated key `FileHeadMasterKey` for our new File Head. We then add a mapping from `filename` to the UUID/Master Key into our `MyFileTable`. We derive the encryption key for the File Head using HKDF on the master key with the string “GetFileHeadEncryptKey”. We upload the updated user struct using GAES with UUID = `FileHeadUUID` and key = encryption key. We create a FileSegment and a random UUID/key for it. We add that segment’s UUID/key to the FileHead’s `SegmentTable` at index 0. We upload the FileHead using GAES. We add the actual file data to the FileSegment and upload that as well. So, here’s the full process:
   a. Generate a random UUID for file’s FileHead, we’ll call it `FileHeadUUID`.
   b. Generate a random key for file’s FileHead, we’ll call it `FileHeadMasterKey`.
   c. From master key derive the key for encrypting the File Head: `FileHeadEncryptKey = HMACEval(MasterKey, "GetFileHeadEncryptKey")[:16]`
   d. Create a new instance of FileHead struct, set `NumSegments` to 1.
   e. Generate a random UUID for this first file segment that will contain the whole file as it is right now, we’ll call it `FileSegmentUUID`.
   f. Generate a random 128-bit key we’ll call `FileSegmentKey.`
   g. Upload file data to Datastore using GAES with UUID = `FileSegmentUUID` and k = `FileSegmentKey.`

h. Add [0 → (`FileSegmentUUID`,`FileSegmentKey`)] to FileHead's `SegmentTable`. (0 because this segment is the first segment, in a zero-indexed context)

i. Upload FileHead to Datastore using GAES with UUID = `FileHeadUUID` and key = `FileHeadEncryptKey`.

j. Add FileHead to user struct's `MyFileTable` as [filename → (`FileHeadUUID, FileHeadMasterKey`)].

k. Upload updated user struct to Datastore using GAES, with UUID = `userdata.UUID` and key = `userdata.K0`.

4. **LoadFile(filename string) (data []byte, err error):**

   We look up the filename in the user's MyFileTable, get the FileHead's UUID and master key, compute the encryption key from the master key, and download the FileHead using GAES. Then we iterate through the FileHead's SegmentTable in order of index., downloading each segment from Datastore using GAES and appending its data to a single string that will be our file. So, here's the full process:

   a. Look for filename in the user's `MyFileTable`. If it exists, get the corresponding `FileHeadUUID` and `FileHeadMasterKey`. From the Master Key, deduce the encryption key and fetch the File Head from Datastore using GAES. We now have a FileHead structure.

   b. Create empty []byte called FileContents. Iterate through the FileHead's `SegmentTable`, from i = 0 → `NumSegments`. Each time, get the UUID at `SegmentTable[i]` → `FileSegmentUUID`. Download this UUID from Datastore using GAES, with key = `SegmentTable[i]` → `FileSegmentKey`. Append data to FileContents.

   c. Return FileContents.

5. **AppendFile(filename string, data []byte) (err error):**

   We will generate a new file segment (essentially a new entry in the Datastore at a random UUID, containing the new data encrypted with a random key). We'll add the new UUID and key to the FileHead's SegmentTable at the next consecutive index. Finally, we upload the new File Segment and updated File Head to Datastore using GAES. So, here's the full process:

   a. Look for filename in the user's `MyFileTable`. If it exists, get the corresponding `FileHeadUUID` and `FileHeadMasterKey`. From the Master Key, deduce the encryption key and fetch the File Head from Datastore using GAES. We now have a FileHead structure.

   b. Generate a random 128-bit key we'll call `NewSegmentKey`.

   c. Generate a random UUID we'll call `NewSegmentUUID`.

   d. Upload new data to Datastore with `NewSegmentUUID` using Auth. Enc. Scheme with k = `NewSegmentKey.`
   e. Add [`NumSegments` ➔ (`NewSegmentUUID`, `NewSegmentKey`)] to FileHead's `SegmentTable`.
   f. Increment FileHead's `NumSegments`.
   g. Upload updated FileHead using GAES with key = `FileHeadEncryptKey`.

6. **ShareFile(filename string, recipient string) (magic_string string, err error)**
   Our 'magic string' is going to be an *encrypted* concatenation of three things: the FileHeadKey, FileHeadUUID, and a randomly generated UUID where we'll store (using GAES) the sender's signature of this magic_string's ciphertext. This signature will be encrypted using a unique key based on the File Head's master key and the two usernames. If successful, the recipient will have the FileHead UUID and the key needed to decrypt it, and can also download the signature on the magic string's cipher and verify that it is from the sender. So, here's the full process:
   a. Query user's `MyFileTable` for the filename. If it exists, get the corresponding `FileHeadUDID` and `FileHeadMasterKey`.
   b. From the Master Key, derive the key for encrypting the signature you will upload to the Datastore. It's uniquely based on filekey and the two usernames involved.
      `FileHeadSignatureKey = HMACEval(MasterKey, userdata.Username + recipient)`
   c. Generate a random UUID `signUUID`, to store the signature in Datastore.
   d. Pre-concatenate this with the other two things, so:

`MagicStr = FileHeadKey || FileHeadUUID || signUUID`

   e. Encrypt `MagicStr` using recipient's Public Asymmetric Encryption Key, we'll call this `MagicEnc`.
   f. Calculate signature on `MagicEnc` using sender's `SignaturePrivKey`, and upload this to the Datastore using GAES with UUID = `signUUID` and key = `FileHeadSignatureKey`.
   g. Return `MagicEnc`.

7. **ReceiveFile(filename string, sender string, magic_string string) error**
   We will decrypt the 'magic string' and break it into the three parts: FileHead UUID, FileHead Master Key, and Signature UUID. We'll derive the signature encryption key from the master key, go get the signature from the Datastore, and verify it on the encrypted magic string using the sender's Public Signing Key. Then we'll add

the FileHeadUUID and FIleHead Master Key to our `MyFileTable`. Upload the updated user struct with GAES. So, here's the full process:

   a. We decrypt the received string `MagicEnc` using our `AsymmPrivKey` to acquire `MagicStr`.
   b. We cut out the `signUUID` and also derive the key for the signature from the Master Key. We download the signature, which we verify on `MagicEnc` using the sender's Public Signing Key.
   c. Cut out the `FileHeadUUID` and `FileHeadMasterKey`, and add to `MyFileTable` the following entry: [`filename`, (`FileHeadUUID`, `FileHeadMasterKey`)].
   d. Upload updated user structure using GAES.

8. **RevokeFile(filename string) error**

   We'll download the File Head and delete it from the Datastore. Then, we upload it to the Datastore with a new random UUID and new random master key, and update our `MyFileTable` to reflect the new UUID/key for that File Head. Now, only we know the updated location/key of the File Head. Old shared users now hold an invalid UUID/master key pair for that file. Upload our updated user struct with GAES. So, here's the full process:

   a. Look for filename in the user's `MyFileTable`. If it exists, get the corresponding `FileHeadUUID` and `FileHeadMasterKey`. From the Master Key, deduce the encryption key and fetch the File Head from Datastore using GAES. We now have a FileHead structure.
   b. Generate a new UUID `NewFileHeadUUID` and a new key `NewFileHeadMasterKey`, which'll be the new details only we'll know.
   c. From `NewFileHeadMasterKey`, generate the new File Head Encryption key `NewFileHeadEncryptKey`.
   d. Upload the File Head struct to the Datastore using GAES with UUID = `NewFileHeadUUID` and key = `NewFileHeadEncryptKey`.
   e. Delete the File Head from the old `FileHeadUUID` in Datastore.
   f. Update user's `MyFileTable` entry for this filename to reflect the new master key and UUID.
   g. Upload the updated user struct using GAES.

Section 2: Security Analysis

Overview

We’ll look at a Dictionary attack, a Birthday attack, a Man-In-The-Middle attack, and a Known Plaintext attack.

Possible Attack Analyses

1. Dictionary Attack (breaking design of user)
   This attack is relevant to how we store and encrypt User structs, because this is the only part of the system deterministic based on the user’s username and password. Usernames have very low entropy (read: predictable) but passwords can be assumed to be strong/have medium-good entropy. An attacker using a dictionary attack would probably try millions of combos of common usernames and passwords to try and get a valid UUID in the Datastore. Therefore, we generate the encryption key for GAES of User structs as such: `K0 = A2K(password, salt = username, 32 bytes)[:16].` By using the predictable username as salt for a password that has medium entropy, Argon2 basically acts as a random permutation and is sufficiently strong against dictionary attacks. The salt helps us because the attacker can’t efficiently calculate the hash of each common password with the salt of each common username. That would be ~O(N^M).

2. Birthday Attack (breaking data storage)
   An attacker could attempt to change an entry in the Datastore and trick GAES into not detecting the change. To do this without detection, he would have to choose an existing entry, and craft a new malicious entry whose ciphertext’s HMAC is equivalent to the existing HMAC of the entry. Luckily, our choice to use HMAC which uses SHA-512 protects us from this, as SHA-512 is extremely collision resistant and it is *extremely* difficult to manually find two inputs that have the same hash. Additionally, he would have to somehow acquire the key used to originally acquire the hash key, which is difficult to the rest of our system’s design.

3. MITM (breaking data transfer between users)
   An attacker could intercept the magic string User A sends to User B to send a file. Remember, the magic string contains the following, encrypted with User B’s public asymmetric key: `FileHeadKey || FileHeadUUID || signUUID`. The attacker can try two things:

   a. First, he could try and interpret the string so he can get access to the shared file. We are protected from this because the string is encrypted using RSA and without User B's private key, breaking this reduces to factoring a large prime.
   b. Secondly, he could try and modify the string to mislead User B, possibly to a malicious file or something else. Three things can happen here:
      i. Simplest case is that he just messes up the string and when User B decrypts, the `signUUID` is just garbage and he fails to retrieve a signature. By design, User B knows something is wrong.
      ii. If he meddles with the string but accidentally leaves the `signUUID` intact, User B will know something went wrong when he downloads the signature stored at `signUUID` and fails to verify the message using User A's public signing key.
      iii. He may try to replace the `signUUID` with the UUID of a signature he uploads of his malicious message, but he doesn't have User A's private signing key so User B will again know that the message is not authentic.

4. Known Plaintext Attack (attacker has access to a message's plaintext and ciphertext)
   An attacker, given the plaintext and ciphertext of a certain message, will try to figure out the encryption key and the MAC key. However, this is almost impossible since, to figure out the MAC key, the attacker would somehow have to reverse HMAC which is a one-way, collision-resistant function. Furthermore, the attacker cannot figure out the Encryption key given the plaintext and its symmetric encryption. The symmetric encryption provides essentially a random permutation (especially with a random IV in CTR mode) and the only way an attacker could figure out the key is if we XORed it with the plaintext (which he has), which our design does not do.