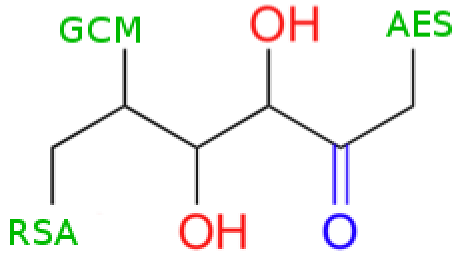
Kryptose™

Security Design: Final Release



kryptose

Personnel: Jonathan Shi (js2845), Antonio Marcedone (am2623), Alexander Guziel (asg252), Jeff Tian (yt336)

Confidentiality/Integrity

For each user, the client creates a data structure (an ArrayList of Credentials) that is used to store the individual credentials that the user wishes to store within our system. The data structure is then serialized, encrypted on the client side and then sent to the server for storage, with key for this encryption never leaving the client device.

The encryption of the stored credentials is done using the AES/GCM/NoPadding ciphersuite. The key is 128 bits long, the length was decided based on NIST guide to key length. The initialization vector for the encryption is randomly generated by the client using a SecureRandom implementation. The GCM mode is an authenticated encryption mode of operation, and ensures that the server is not able to modify (or even read) the stored credentials, as well as any metadata that are stored with them. Therefore a malicious server could only re-play to the client an older version of the credential list, but not tamper it in any way.

The integrity and confidentiality of the connection between the client and the server are guaranteed by the use of SSL connections. In particular, we only accept the latest TLS1.2 version of the protocol, with either the TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 or the TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 ciphersuites. Diffie Hellman Key Exchange (or its elliptic curve version) provide perfect forward secrecy, and a strong block cipher and hash function (with an authenticated encryption mode of operation) guarantee a strong level of security. No vulnerabilities on this version of TLS and ciphersuites are known at the time of writing.

The logs are encrypted in a tamper-proof way with an administrator password that is not kept on the server machine, hence enforcing confidentiality of logged information from attackers.

Audit

We log every connection and user request on the server side, and these logs are viewable only by the server administrator (at the moment). There are two types of logs: general server logs and user-specific logs.

The server logs contain all of the major actions and exceptions raised and caught by the server, along with a timestamp for each. This includes when the server boots up, what port it listens on, which SSL connection requests it receives and accepts, and the exceptions it throws internally and handles. This log is stored in XML format and is generated by the java.util.logging library in Java. We also encode the log in tamper-proof format. The key is derived with PBKDF2 with an application specific salt from an administrator password. There is a utility to read the logs and to set up the secret key for tamper proof logging.

On the user-specific client logs, the server logs any operation that the client requests, and whether the it is fulfilled or fails. For example, a successful put request where the client stores a new version of the password file, and a successful get request when a client retrieves their password file, are both logged. Each log entry contains the date and time the request was received, the username of the user who initiated the action, the type of request, and whether or not the request was successful. If the request was not successful, the log also contains the reason for the request failure, for example a get request could fail because of invalid client credentials, and a put request could fail because of a stale write issue. These logs are displayed in a user-friendly format, and the intention is that by the final release the user should be able to request to see a copy of their own log to audit their account.

Authentication

The server authenticates the client using a secret authentication key derived from the same master password that is used to derive the encryption key for the credentials. This authentication key is not to be stored by the server as it is, but is hashed and salted (using PBKDF2, which is NIST approved and designed to be slow and therefore resistant to brute-force attacks). To authenticate a user, the server repeats the computation and authentication succeeds if the recomputed value equals what was stored in memory. The salt used is generated independently and randomly for each user, and is changed every time the user changes the master password (and therefore the authentication key).

When the user creates an account, this hashed and salted version of the authentication password is associated with the user for the first time in the server and stored permanently. All subsequent requests by the client (the request design pattern is explained in more detail in the authentication portion of the document) must contain the client authentication key, and with each request the client is authenticated when the server validates the key in the manner described in the previous paragraph. When a client changes their master password, this action is a type of request, so the client must authenticate the change password request with their old authentication password, and when this request is approved by the reference monitor, the hashed and salted password associated with the user is replaced by the hashed and salted version of the new password.

The client authenticates the server by using digital certificates. We created our own certification authority and installed its certificate on the client. We provided the server with a signed certificate. The client DOES NOT perform hostname validation at the moment, meaning that any certificate signed by our CA will be accepted as valid by the client, without consideration for the common name on the certificate.

This is not a concern, as we assume that our CA will only issue certificates to our own server for now. We decided for this approach (as opposed to installing a simple self signed certificate on the client) to allow the use of short validity certificates and thus limit the window of time for a MITM attack in case of server compromise. This will also make the switch to certificates signed by a real certification authority easier.

We might add hostname validation and possibly certificate revocation lists in the final release.

Multiple security elements

For each account, both the key used to encrypt the data and the authentication key are derived from the same master password, again using the PBKDF2 function. The function supports a variable length output, so we split it into two chunks and derive two keys. The security specification guarantees that the bits are pseudorandom and therefore knowledge of the authentication key does not help an adversary in decrypting the data. For the derivation of these two keys, instead of using a per-user salt (which we do not do since we are in part protecting against the server as an adversary and we want to keep stateless clients), we use a value of the salt which is set per application, but include the username together with the password as an input to the algorithm. This ensures that lookup (or rainbow) tables need to be built for each individual user, and therefore offer no advantage over a per-user brute force attack.

Authorization

We used a mandatory access control policy (MAC). This can be seen in how our central security policy administrator (the reference monitor of the server) completely controls all abilities in the system; subjects cannot delegate permissions to each other as they would be able to under a discretionary access control policy.

The way the authentication portion of our system works is that the client and server do not maintain any explicit session when the client logs in. Instead, every time the client makes an action that requires talking to the server (such as saving a new credential that needs to be synced with the server), the client generates a *request*, which encodes the action the server should take, along with the username and the derived authentication key of the client. Upon receiving such a request, the *reference monitor* of the server first checks that the authentication password in the request matches the expected authentication password of the subject which the request is for. Next, it checks that the request is indeed a valid one that encodes an action that the subject is indeed allowed to perform under our access control policy. The server proceeds with the request only if both of these checks pass as determined by the reference monitor.

As can be seen by our system design, the abilities of subjects are completely determined by this central reference monitor, there is no additional infrastructure in place for subjects to delegate certain abilities to one another. To do so would entail that a subject gives their authentication key to another subject, which is obviously not a valid mechanism because as in any system, subjects must never share their authentication credentials with any other subject.

The types of abilities for each subject are pretty uniform, in the sense that since there is only one user type in our system (which is just a regular Kryptose user, we have an admin role that is somewhat outside of our system that we will discuss in the following parts) and due to the nature of the system we are building (since our system is a password manager it doesn't make sense for users to share credentials with each other, as opposed to say a filesystem where sharing would make a lot more sense), each user has the same set of abilities for changing their own account, and no abilities for anything relating to other users' accounts or the audit logs. A user's abilities include retrieving their credentials, writing their credentials, changing their master password, creating a new account, and deleting an existing account. We set up a request type for each of these actions. So the reference monitor approves each of these request types from clients, provided the given authentication password is correct. Other potential abilities we don't want (such as a user reading server logs) are precluded because there simply does not exist a request for these abilities.

To change the authorization policy, one can simply add a request type to give users a new ability, or delete the request type to remove an ability. Again due to the nature of the system there is no foreseeable need to allow for different clients to each have a different per-client dynamic set of abilities to be delegated and revoked continuously.

In addition, there is a sys admin role that is not really part of the system per-se, the admin is simply the server administrator who sets up and runs the server part of Kryptose on a server machine and sets up the initial key for the tamper proof logs for the hash chain to be initialized. This makes the sys admin the sole principal with the ability to read the audit logs, although this ability is outside of the periphery controlled by our access control monitor, it simply involves the sys admin decrypting the log files with the initial key (we provide standalone code for this decryption).