Technical facts about zkBox

… or put it in another way: I'm a nerd! Tell me more about the architecture and what's underneath

Deployment types

- there is a publicly available installation of zkBox deployed in the Amazon Cloud (running on EC2 instances) and using Amazon SimpleDB and Amazon S3 as backend: http://api.zkbox.com
- but you can have your own installation of zkBox by either storing the data in the Amazon S3+SimpleDB or in a SQL Server database (e.g. if your policy doesn't allow to have data stored outside of company's perimeter)

Usage

- can be seen as a privacy-as-a-service solution by offering an encrypted storage in a NoSQL online database with a zero-knowledge authentication mechanism on top
- the interaction with the zkBox web service is performed via a **REST+JSON** API
- if you are developing a web application:
  - you need to include the WebProxy in your solution; the proxy is necessary due to the browser's security limitations: there is not way to make direct Ajax calls from the client browser to the API
  - in your pages include the JavaScript API; you'll call the zkBox methods that will communicate with the proxy and will handle everything for you
- if you are developing a desktop application
  - you need to include the API client in your solution; you'll make the calls to the API which will handle everything for you
  - on the backed, the API can store the data in Amazon S3+SimpleDB or in a SQL Server database

Structure

- the structure of the zkBox data is very simple; there are 3 types of entities:
  - **Applications**: You are requesting an (applicationId, applicationKey) pair. You are using them to initialize the WebProxy (if on a web application) or the API client (if on a desktop application). The applicationKey is used to sign the calls to the storage; its never sent out over the wire.
  - **Users**: Each application can create its own users (a user is identified by an userId). For each user there is a verifier stored that is used to authenticate the user (using the zero-knowledge proof algorithm SRP variant 6a). On the client there is a encryption key derived from its login that is never sent out.
  - **Objects**: Each user have access to its collection of objects after authentication. An object is identified by an objectId and it has a type (which is stored unencrypted to allow selective storage queries) a the data (which is always encrypted and decrypted by the client).

Algorithms used

- the **SRP 6a** is used for authentication; it's a secure password-based authentication and key-exchange protocol
- for hashing, the **SHA512** and **HMAC512** algorithms are used
- for encryption, the **AES256** algorithm is implemented(with a 64bit salt)
- a true random number generator (**TRNG**) is used (the randomness comes from atmospheric noise) with a fallback to a pseudo random number generator (PRNG) to prevent as much as possible the prediction

Security considerations

- each call to the storage (including the authentication steps) are coming with a different identifier (nonce) which is stored and checked for further calls; in this way the reply attacks are prevented
- the encryption key is derived from the password by performing a big number of cascaded HMAC operations to harden the password computation in the unlikely event that an attacker decrypt a piece of data
- every hashing and encryption is salted and prefixed with a constant to prevent dictionary based or rainbow tables attacks
- for applications, only the applicationId travels around, the applicationKey is never sent out from the application server
- for the users, only the user login travels in hashed form and the userId; the password or the encryption key is never sent out from the client machine
- for objects, only the objectId and the object type is traveling unencrypted; the object data is never sent out unencrypted from the client machine
- all the ids are generated pseudo-randomly so there is no information gained just by looking at ids
- the long sessions identifiers are preventing guessing them
- delays on failed authentications or invalid requests to prevent brute force attacks
- in-memory cookie on the client to prevent session hijacking
- even if some malicious person gains access to the data, it will not be able to do anything since all the data is encrypted; more over it's encrypted with different keys, since each user have it's own encryption key
- as an extra security level, all the communication is performed over SSL only
- the user's data is double signed: once by the user and once by the application; in this way not even the application owner (besides the zkBox staff) is able to peak or alter user's data
- total anonymity: the data is never leaving the client's machine in clear form; only the owner of the data can read it; there is no profile information required when requesting to add new applications or new users to the storage

Performance

- zkBox was designed having the scalability in mind and avoiding having single points of failure (SPOF)
- a distributed load balancer is placed in front of the API instances
- multiple API instance can be spawned according to the load
- the distributed cache layer ensure fast response for frequently accessed objects and solves the eventual consistency when using Amazon Services as the backed
- there are two types of nodes: cache nodes and zkBox API nodes; any node can fail and the load is taken by other instances
- there is no session stickiness and the sessions are stored in the data model layer, so any failure of a node will not affect the clients
- smart caching, heavy usage of parallelization (especially when working with the Amazon storages) and asynchronous storage requests are used to achieve a minimum response time