Mitro: Password Manager for Groups/Enterprise

Mitro is a password manager that supports groups and enterprises. It integrates with web browsers to generate and save passwords, meaning that users only need one strong password to sign in to any web application. It also supports sharing passwords with other individuals or groups, and supports “organizations” that can apply their own policies.

Use Cases

This section describes how people will use the tool, without discussing any specific interface or technical requirements.

Individual

Wants something to generate/remember personal passwords across all browsers and devices. E.g. bank account login, email login, Twitter account.

- Logs in to a web site with an existing password: prompts to save password.
- Creates a new account on a site: prompt to generate a unique password.
- Visits a site on which he/she has an account: prompts to sign in / autofill.
- Changes a password on a site: we save the new password. We also save all previous versions, so in case we/they screw up, the user can find their old password.
- Can choose to copy or view the password, although we discourage it in the UI. This is needed for non-web logins.
- Saves secure notes (e.g. credit card numbers, database server passwords, root certificates, etc.)
- Accesses their passwords on multiple devices (web browsers, mobile, etc).
- Optional: If they forget their password, it should be possible to reset it. Suggestion options: Escrow using an offline key with us (manual recovery), downloadable “recovery code file”, store a Google Doc/Drive/Dropbox recovery file (via the JS API so it is done on the client side).

Two Factor Authentication

Some users will want to enable two factor authentication (2fa) for extra security.

- User selects the “Two Factor Auth Preferences” in the popup, and a tab opens up to the Two Factor Authentication Preferences page.
- The page either:
  a. Says the user has enabled 2fa and lets the user disable it or reset the backup codes.
  b. Says the user hasn’t enabled 2fa, explains what it is and lets the user enable it.
- Enable 2fa: prompt to scan the QR code and input the verification code from the app. If the code was correct: backup codes are shown. If not: prompt to input code again.
- Login on a new device: if 2fa is enabled, prompt to input verification or backup code
- Change password: if 2fa is enabled, prompt to input verification or backup code

Sharing

Some accounts need to be shared, for both personal reasons (e.g. sharing a bank account with a spouse) and for work (e.g. shared Twitter account, shared credit card, hiring a contractor).

- User selects an account or secure note and chooses people/groups to share it with. People are identified by email address, so we can include people who don’t already have accounts. This password then appears in the same way as all other passwords.
- Users are either “owners” or “users.” Owners can control who has access and edit the metadata. Regular users can only use the password. *NOTE*: This is more for convenience, not for security: anyone who has access to the password can copy it and change the password on the service. This is a convenience for mostly trusted teams.
- When revoking access, if the user being removed has ever accessed the password, we

suggest changing it (to guarantee that they no longer have access).
- Users can create groups of people and passwords. e.g. digital agencies can create a group per client, then assign users and passwords to that client as appropriate. Groups also have access levels of “owners” and “users”.

Organizations

Organizations want additional control: administrator access to all passwords, limits on sharing (e.g. forbid sharing externally), policies about authentication (e.g. timeouts, 2FA), resetting user passwords, etc. They may additionally wish to synchronize with existing directories (e.g. Google Apps, Active Directory, LDAP). Directories may have a hierarchy of “organizational units” in addition to groups. However, for users it should work the same: they save passwords in an organization, and can use them as normal.

- An organization’s administrators can access and edit anything that belongs to an organization (accounts/secure notes, and groups). This way they can see what services their organization is using, can access accounts if someone leaves, and can reset passwords.
- An organization’s members can create new accounts/secure notes and new groups as part of the organization.
- Users can use both their personal passwords and organization’s passwords at the same time, from the same browser.
- When saving a new password the client asks what organization it should belong to (e.g. Org A, Org B, Personal), defaulting to whatever was selected last.
- Users can save individual accounts/secure notes that belong to the organization. E.g. their own Salesforce, YouSendIt or DNS accounts. These are accessible only to them, and to the organization administrators.
- Sharing an account/secure note with other members of the organization works as normal (as permitted by the ACLs of each secret/group)
- When sharing a password with someone who is not a member of the organization, the company policy applies. By default, it will display a warning, but allow anyone to invite new users (similar to Google Docs). Other policies are “deny” or “ask administrator”.
- Audit/access logs: Administrators can see who has accessed which passwords, from which devices.
- Device/usage reports: Administrators can see which devices have been used to access the service.
- Backups and exporting: For disaster recovery and to avoid lock in.

Enterprise features we may want to support in the future

- Enforce rules about how users authenticate. E.g. must type a password once a day, use two-factor authentication, or other policies.
- Store an organization’s passwords on their own server, while still using passwords on

the “public” server (e.g. personal).
- Restrict access to the organization’s passwords to specific devices.
- Force all passwords to belong to the organization (e.g. a bank, on a bank owned machine).
- Restrict users so they can’t create their own groups.
- Allow users to see all passwords in the organization, and allow them to request access. One agency likes this feature of their existing system, since it permits a level of self-service access, while allowing managers to control who has access to what.

Core Cryptographic Design

Mitro is divided into two parts: the core service and the user interface. The core service is security critical and performs all the cryptographic operations. It is intended to be as simple as possible to make it easier to verify the design and the implementation. The user interface maps user operations to simpler core service operations. The user interface is also inevitably involved in maintaining security. However, if the core service is correct, flaws in the user interface should only have limited impact. Functionality in both parts is divided between clients and the server.

Mitro Core is composed of a client, typically running as a web browser extension, and the *keyserver*. The server is trusted as little as possible, since it will be run as a third party service and can be easily attacked. The core service stores three types of data: **identities**, **groups**, and **secrets**. Identities are public/private keys that represent users. Groups implement access control lists to limit who can see, access and edit secrets. Secrets store usernames/passwords or other secret data.

Identities

To use the service, a client must create an *identity*. An identity is composed of a unique user id (usually an email address), a public key, an encrypted private key, a display name, and additional directory data. The client code generates the key pair, and encrypts the private key using a strong passphrase. **The security of the system relies on the fact that only this authorized user can access the decrypted secret key**.

To create an identity, the client submits a request to a server with a *proof of identity*. This proof is an assertion from a trusted third party that the user submitting the request really controls the user id (e.g. a Google OAuth token, or a token from our own email verification service). The keyserver verifies the token, then signs the identity record with its own private key. This signature proves to clients that the keyserver verified the identity, and means someone with access to the database cannot insert or modify identities.

Identities store:

- user id
- name
- public key
- encrypted private key
- additional optional directory data (e.g. profile picture, web site, contact details, ...)
- keyserver signature on (user id, public key)
- two factor authentication secret and backup codes

Groups

Groups apply access control policies to a set of secrets (see below). Groups contain members in an Access Control List (ACL), each of which in an identity or another groups. Each group also has its own public/private key pair. The private key is individually encrypted for each group member, so only that member can decrypt it. Each ACL entry is signed with the group key, so the keyserver and clients can verify that it was added by a member of the group (otherwise, someone might add themselves to the group, and the next time the group key was changed they would get access).

When adding a secret to a group, the secret is encrypted with the group public key, so only members of the group can access it. When removing a user, the group key needs to be rotated, to ensure that if an attacker gets access to the encrypted data, previous members cannot access it. The secret record is signed by a group member so that clients and the keyserver can prove that a valid member of the group added it, and that someone is not attempting to inject a record for a group.

Group members have two levels of access: *owner* or *member*. Owners can edit everything inside a group: the group metadata, add/remove secrets, and changing the membership. Members can only read the group metadata, secrets and membership list. Since groups can contain other groups, this provides a flexible graph relationship that can represent both hierarchies and cross-organizational groupings. We never permit cycles in the group graph, which is enforced when adding groups as owners/members. Each group must also have at least one owner.

Identities can see the public metadata for all groups that they belong to as an owner/member, as well as all the groups that are included in those groups as owners/members.

Group data:

- group unique id
- name / label
- public key
- group key signature (name, public key)
- list of ACL entries
- list of contained secrets, encrypted for this group

ACL data:

- acl unique id
- group id (pointer to group, above)
- level (member or owner)
- member group id (can be null)
- member identity id (can be null)
- encrypted group private key for the group/identity
- group key signature (parent group id, level, member group id, identity id)

Secret data:

- parent group id
- secret unique id
- secret version id
- client visible data (encrypted with group key)
- critical data (encrypted with group key)
- group key signature (parent group id, secret unit id)

ACL Signatures

Signatures on ACLs are required to prevent an attacker with write access to the database from injecting themselves into a group, as follows:

1. Attacker adds their identity to a large group, where it is unlikely to be noticed. At this point, they cannot access any secrets, because they do not have access to the group key.
2. An administrator removes a normal user from the group. This causes the group key to be rotated. As part of this operation, they generate a new group key, and encrypt it for all the members, *including Attacker*.
3. Attacker now has access to the secrets in the group.

Signing each ACL entry with the group key itself solves part of the problem: now in step 2, the administrator will see that Attacker's entry does not have a valid signature and detect the problem. Even better, Mitro can periodically scan the database for suspicious data. However, it does not protect from an attacker spoofing the entire group:

1. Attacker replaces the target group's key with a new generated key.
2. Attacker updates all ACL entries to be signed with the new key, and adds their own entry.
3. Attacker rewrites all secrets to generated fake data they have created.
4. A user or administrator saves a new password to the group. Or, they see that a password no longer works and reset it / resave it.
5. The attacker now has access to that password.

This requires users to not notice that the secrets have changed or been deleted, which is less likely but not impossible. TODO: Can we protect against this in an intelligent way?

Access Control Example

Group Engineering: members: Id 1; owners: Id 2, Group Company
Group Sales: members: Id 3; owners: Id 4, Group Company
Group Company: members: Id 5; owners: Id 6
Group All Staff: members: Group Engineering, Group Sales; owners: Group Company

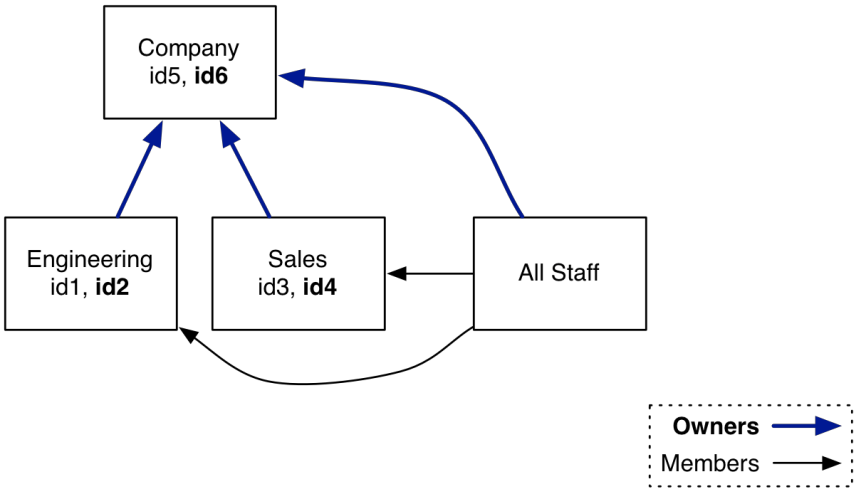

Figure 1: Membership graph, arrows point to included groups

In the figure above, arrows point from a group to included groups. Access to secrets flows along these edges, meaning that secrets in “upstream” groups can be accessed by identities in the “downstream” group(s). In the example, secrets in Group Engineering can be accessed by identities 1, 2, 5 and 6. They can be edited by identities 2 (e.g. an engineering manager), 5 and 6. The company group can only be edited by identity 6 (e.g. the CEO), although identity 5 can edit anything in Sales and Engineering.

Company, Engineering and Sales form a strict hierarchy with Company at the top level. All Staff is a “cross hierarchy” group (e.g. global engineering staff, containing engineers from both Canada and the United States). Secrets in All Staff are visible to everyone, but can only be administered by people in the top-level Company group.

When identity 1 lists their groups, they see Engineering and All Staff (because they are members), and Company (because it is included in Engineering). They cannot see the Sales group. Identity 6 can see all the groups, since they are an owner of all of them.

Secrets

Secrets store encrypted data. Every secret belongs to at least one group that controls who can

access it. The secret has three parts: server visible, client visible, and security critical. The server visible part is visible to the server, for operational reasons. The client visible part is encrypted, but returned with the “list secrets” operation, and is displayed to the user. Hence, this part should not contain critical data. The security critical part is only returned when this secret is accessed individually. The encryption ensures that only members of the group(s) this secret belongs can access to it. Secrets are versioned, so previous revisions of the data (e.g. username, password) are still accessible. This allows users to recover from accidental edits. The system will automatically delete old revisions after 30 days, and users can explicitly expunge previous revisions. This allows users to permanently remove data that was accidentally saved.

Server-visible record:

- secret id
- version id
- edit timestamp
- url / hostname (so Mitro can improve the service to work with these web sites. **TODO:** allow clients to opt out and include this field in the client visible section?)

Secret data (actually stored as part of the group record):

- name / label (client visible)
- long description (client visible; optional)
- user name (client visible)
- password **(critical)**
- secret data **(critical)**

Secret storage example

We will walk through an example of how the encryption of groups/identities work. The simplest case is a single private password. In this case, identity U creates a new Group X to store the ACL for the password. The user generates a new key pair for the group, and encrypts it with their public key so only identity U can decrypt it. They create the following record(s) for the group with its ACL:

```
Group: {
        “id”: (server assigned Group X id),
        “public_key”: “(group X’s public key)”,
        “acl”: [{
                “member_identity”: “U@example.com”,
                “access_level”: “OWNER”,
                “encrypted_private_key”: “(group X’s private key, encrypted for U)”,
        }]
}
```

They then encrypt the secret with this group key, creating these records for the Secret and its

group ownership:

```
Secret: {
        “id”: (server assigned secret id),
        “metadata”: “server-visible metadata”,
        “groups”: [{
                “id”: (server assigned Group X id),
                “client_visible_data”: (client visible data encrypted with Group X’s key),
                “critical_data”: (critical data encrypted with Group X’s key)
        }]
}
```

Now, if identity U wishes to decrypt this secret, they must perform the following operations:

1. Find the shortest access path from the secret to their identity (Secret -> Group X -> Identity U). The data is decrypted in the reverse order of this path as follows.
2. Decrypt the Group X key with Identity U’s private key.
3. Decrypt the secret with the Group X key.

To give an additional identity V access to this secret, we can add them to Group X. To do this, the client must re-encrypt the group private key for identity V, creating this record:

```
Group: {
        “id”: (server assigned Group X id),
        “public_key”: “(group X’s public key)”,
        “acl”: [{
                “member_identity”: “U@example.com”,
                “access_level”: “OWNER”,
                “encrypted_private_key”: “(group X’s private key, encrypted for U)”,
        }, {
                “member_identity”: “V@example.com”,
                “access_level”: “OWNER”,
                “encrypted_private_key”: “(group X’s private key, encrypted for V)”,
        }]
}
```

This gives identity V access to everything in Group X (a single secret).

Discarded Design Alternatives/Discussion

Encrypt secrets directly for users (no group keys): This could eliminate the need for groups and group keys. Instead, secrets would be encrypted for the “flattened” group membership. This

is perfect for “private” secrets accessible to one user, or for secrets shared with a small group. However, it doesn’t scale as well to large groups. Considering the number of operations required (below), all operations except accessing a secret and revoking a user’s access are cheaper with group keys. The more expensive operations are only slightly more expensive.

U = # users in the group; S = # secrets in the group

No group keys:
Access a secret: decrypt the secret with the user’s decrypted key (1)
Add secret to group: encrypt secret once per user in group (U)
Remove secret from group: Delete ACL entries for all users (U)
Add user to group: re-encrypt all the group’s secrets for that user (S)
Remove user from group: Delete ACL entries (S)

Group keys:
Access a secret: decrypt the group key then the secret (1 ; but technically 2 operations)
Add secret to group: encrypt secret with group key (1)
Remove secret from group: Delete ACL entry (1)
Add user to group: Encrypt the group’s private key for that user (1)
Remove user from group: Delete ACL entry, re-encrypt secrets with new group key (1 + S)

Flat groups (no nesting): this would simplify ACLs. However, we would need to represent organizations in some way, to ensure that administrators have access to organizational secrets. This is now represented as nested groups. This also can’t directly represent more complex directory structures, which we may need in the future.

Symmetric group keys: Group encryption keys are asymmetric, allowing users in the group to sign things and permitting the keyserver to verify them. It also could permit people who are not members of the group to add secrets to the group (currently not supported). For example, consider a user in the Marketing group, who wants to share a password for an analytics account with the Engineering group, but who does not belong to the Engineering group.

Using symmetric keys for groups would permit more efficient encryption/decryption. In particular, asymmetric crypto only be used directly for short data inputs (< ~200 bytes with OAEP). For longer inputs, it is typically used to encrypt a random symmetric key, then that key is used to encrypt the data. However, users only rarely access secrets, so this shouldn’t be a problem.

No group access level (containment only): Instead of assigning groups as members or owners, we could just support containment. (e.g. Company contains Engineering and Sales). In this version, admins of the “containing” group are automatically admins for the “contained” group.

This seems fine for users, but doesn't work so well for secrets. Consider the example of creating an All Staff group that has globally accessible secrets, but is administered by a few users. This can almost be represented by having Engineering, Sales, and Company all contain All Staff, but then all the admins in Engineering, Sales, and Company can edit the secrets. Fixing this requires adding access levels, which gets us back to our current system.

Implementation

The server is written as a Java servlet (using Jetty), as this is widely understood technology, making it easy for people to deploy it themselves, or run it on PaaS systems like Heroku or Google Appengine. The client is written in Javascript, as it must run on multiple web browsers as an extension (and possibly as part of a plain web page). Javascript extensions' background tasks run in a completely isolated environment from web pages and from other extensions.

For cryptography, we use Google's Keyczar. This library is designed to make it hard to make mistakes when using low-level cryptographic primitives. It is also released under a commercial-friendly licence (Apache), and has C++ and Python versions. All secrets will be encrypted using Keyczar's "session encryption", which generates a random symmetric key, encrypts the data with that key, then encrypts the symmetric key with the asymmetric key. While this is inefficient for small data ~< 200 bytes (e.g. most passwords), it means we can support large secrets and only audit a single code path. Most critically, we will need to write an implementation of the subset of Keyczar that we use for Javascript. Keyczar currently uses 4096-bit RSA keys in OAEP mode for public/private key encryption, 4096-bit RSA key for public/private key signing (separate from encryption keys), and 128-bit AES in CBC mode for symmetric encryption. To encrypt identity private keys, it generates a 128-bit AES key from a password using PBKDF2. (**TODO**: Alternatives are NaCl/libsodium, which uses non-NIST primitives by default, or OpenPGP implemented by GPG and Bouncy Castle; is Keyczar the best choice?)

Client/Server Communication

All client/server communication occurs over HTTPS. The client always verifies that the server's SSL certificate is a specific one, configured when the client is built. We rely on automatic updates to change the certificate when it is about to expire. All modification requests are HTTP POSTs, while reads are HTTP GETs. Each request and response is a JSON object with string keys ({"k1:" some value; "k2": other value}). To prove identity, the client serializes the request to a string, and signs it. The actual request sent to the server is:

{"identity": "unique identity string", "request": "**... serialized JSON data ...**", "signature": "web safe base64-encoded signature"}

The server verifies the signature using the identity's public key. If it matches, it then processes the request. If not, it returns an HTTP 401 Unauthorized error. This simple protocol is vulnerable

to replay attacks, but SSL protects against that. (**TODO**: Is there some simple existing protocol we should be reusing here? Should we just do our own encryption and signing?)

The one exception is the Create Identity request, which is self-signed (to prove that the client actually has the private key), and the Get Private Key request, which is not signed.

Client Storage

Clients cache their identities locally. The private key MUST be stored encrypted. On first use, it asks for the password to decrypt the key. It then caches the decrypted key in memory (according to policy: after a time limit it drops it). (**TODO**: what countermeasures can we take to make it difficult to attack this key? E.g. swappability, protection from debugging tools or other processes?)

Clients do not cache encrypted secrets locally, and zero decrypted data from memory ASAP (not actually possible in JS, but possible with NaCL and mobile apps). This makes it easier to revoke shared secrets, provides an additional level of protection (a stolen laptop has limited information), and a level of auditability (the server logs are more accurate). It has the downside that when the service is unavailable, so are the secrets. Clients can cache public keys for groups and identities.

Operations

- **Create Identity**: Client generates a new identity. The server verifies and stores it. Requires an external proof of identity.
- **Edit Identity**: Edit the metadata on an identity.
- **Rotate Identity Private Key**: The client provides a new key, and *all* their ACLs and secrets re-encrypted with this key. Useful if a laptop or device is lost/stolen. For organizations, this can be performed by an administrator. For individuals, only by themselves. **TODO**: For individuals, do we need protection against malicious resets?
- **Delete Identity**: Remove the identity and any groups it belongs to. Remove all empty groups. Remove all secrets that become orphans, or reassign ownership to an organization administrator.
- **Get Identity**: Returns the public key and metadata for the identity.
- **Get Private Key**: Returns the encrypted private key for an identity. Requires an external proof of identity. Used to synchronize new devices.

- **Create Group**: Client generates a new group key, encrypts it for themselves and for anyone else in the ACL. The server stores the group.
- **Edit Group**: Edit the metadata and optionally the ACL on the group. When revoking access to a group or a user, the group key should be regenerated, so this needs to include re-encrypted versions of secrets that are part of this group. To delete a group: remove all members.

- **Get Group Private Key**: Returns the metadata for a group, the group's ACL, and the entire ACL path from the identity to the group, so they can actually decrypt the private key.
- **List Groups**: Returns the metadata and public key for all groups this identity has access to: the entire connected component of the group ACL graph.

- **Create Secret**: Client encrypts a secret for at least one group.
- **Edit Secret**: Client creates a new version of a secret and/or changes the ACL. When changing the ACL, it must re-encrypt the version history. To delete a secret: remove it from all groups.
- **Get Secret**: Return the critical part of a secret, and the ACL path to the secret, if the identity has access. This returns all versions of the secret.
- **List Secrets**: Returns the server-visible and client-visible components of all secrets this identity has access to. It also returns the public part of the ACL path to each secret (not the encrypted keys!).

Synchronizing Groups with External Directories

There are two external directories that our early users are interested in using with Mitro: LDAP (including Active Directory) and Google Apps. Users want to use groups stored in the external directory to control access to secrets with Mitro.

Adding users to a group requires administrator credentials to be able to give the new user access. This means either customers must run the agent, or must at least approve the changes through their own client to ensure that Mitro does not have access to the decrypted data. Removing users is easier, since we can remove the ACL entries (however we still need to rotate the keys). A tricky case is when the removed user is the sole owner of some passwords. In the proposed structure for organizations, administrators will always be an implicit, hidden "owner" for all secrets, so the ownership can be "reassigned" without needing any cryptographic operations. However, it may still be desirable to have an explicit "reassignment" of these secrets.

Customer-managed synchronization agent

In this mode, the customer runs a daemon (provided by us) that polls the external directory and mirrors the groups in Mitro. This daemon has access to a "role account" that is an administrator for all the groups and secrets in an organization. It needs this level of access to be able to give new users access to secrets. The customer must run this agent and ensure it is secure.

Mitro-hosted group syncing, with administrator approval

In this mode, Mitro polls the external directory. When changes are detected, it records the desired changes (add user, remove user, etc), then notifies the administrator(s) of the organization via email. The administrator then logs in to Mitro and is shown the changes that

need to be applied. They click “approve changes,” and then their client makes the changes, using their identity. In this mode, Mitro does not have access to any secrets. However, administrators must examine the group membership changes to ensure that a malicious attacker is not trying to add an unauthorized user to the groups.

Mitro-hosted group syncing with automatic changes

In this mode, Mitro runs a daemon that automatically applies changes to the groups. This daemon must have access to the secrets, and will be sandboxed to the highest extent possible. We can even require an administrator to “start” this daemon, after which we do not have any access to it except through the API. This means manual action would be required after a failure, but it provides additional protection against attackers gaining access to secrets.

The User-Visible Model

Mitro exposes a friendlier model to users, built on top of the core. This model is similar to GitHub. The client still performs the core cryptographic operations, but it communicates with a frontend server that helps implement these higher level primitives.

Users/personal identity: Users create a personal identity by verifying their email address (either via Google OAuth or a clicking a link in a confirmation email). This is intended to belong to the person creating the account, not any organizations. **TODO**: Allow multiple email addresses, and changing the primary email. Allow a user to have multiple identities, so that an organization can impose its own key policies on users, while permitting users to still use Mitro for their personal accounts.

Private passwords: When creating a personal password, it is represented by creating a new “hidden” group with the creator as an owner, then storing the password in that group.

Ad-hoc sharing: Users can edit access control lists on their personal secrets. We translate this into editing the “hidden” group containing the secret.

Inviting new users: If a user invites someone who does not yet have an identity, we create an identity for them, and mark it as “unclaimed”. This means we generate a public/private key, and store the private key unencrypted. The user claims the identity, by proving that they have access to the email address. At this point we “release” the private key to them. **TODO**: Currently we send the user an email with a temporary password, then expunge the password from our system. This is good in that we don’t have the key for very long, but bad because if the user can’t find that message they can’t get their account.

Flat groups: Individuals can create groups, and assign passwords to groups. We do not allow nesting, and only allow users to belong to groups. This is because regular mortals do not understand nested groups. TODO: Is this too limiting for organizations/enterprises?

Organizations: Organizations apply administrative policy to a set of identities, groups, and secrets. Organizations are represented by a new group owned by the administrators of the organization. This group is tagged to indicate that it is an organization (otherwise it looks the same as an ad-hoc group of users). All groups in the organization are owned by this administrator group, which gives the administrators access to everything in the organization.

There are three levels of organization membership:

- **Organization owners**: Have full read/write administrative access to all passwords, groups, and ACLs in the organization. However, the UI separates the passwords they have *direct* access to (as "regular" users), and the ones they only have access to as administrators. They must explicitly go to the admin interface to see/edit the others. This triggers some manual re-authentication (e.g. retype your password or 2 factor auth). (**TODO**: Should we generate separate identities, which means administrative operations could be protected differently e.g. requiring re-authorization? If they have separate IDs, administrators probably shouldn't be able to create secrets, but only manage them)
- **Organization members**: Can view their own passwords, save new individual passwords in the organization, and can edit the groups as permitted by their ownership level.
- **Organization guests**: Users that are not part of the organization can be invited to access a group or password by an existing member (if permitted by policy). Guests are not permitted to store new secrets in the organization, but only access what has been explicitly shared with them. This allows limited sharing with external people (e.g. contractors).

The primary user-visible part of an organization is when saving new passwords, they have to select if it belongs to the organization or not. If the user has not configured a "personal" account, everything belongs to the organization.

For each member of the organization (including administrators), we create a "hidden" group with that member as an owner. This allows members to store "private" passwords that belong to the organization, and share them with others in an ad-hoc fashion (just like they do normally with the service for their private passwords).

Two Factor Authentication---How it works

The user goes to /mitro-core/TwoFactorAuth/TwoFactorPreferences by pressing a link in the popup page. When the user does that, a token signed by the user and containing the user's email address and a nonce is also sent to the 2fa preferences page. The preferences page checks the token and signature combination, and then the page determines whether 2fa is enabled or disabled by checking whether identity.getTwoFactorSecret() is null. If it is disabled, information about 2fa is displayed, and the user can press an "Enable" button. This redirect to mitro-core/TwoFactorAuth/NewUser and also passes in the token and signature, both of which are then checked again. Instructions on how to enable 2fa are displayed, along with a QR code

which embeds both the email and a randomly generated “secret” in it, and a form to input the verification code. After entering the code, the user is redirected to mitro-core/TwoFactorAuth/TempVerify and passes along the token, signature, secret, and verification code. The code is checked, and if it is correct, the secret is stored in the database and backup codes are shown (the user may only see them at this point because they are one way hashed). Whenever the user goes to the preferences page after that, it will display that 2fa is enabled, the date it was enabled, how to disable it, as well as how to generate new backup codes.
When 2fa is enabled and a user signs in from a new device, GetMyPrivateKey will through a DoTwoFactorException. The rawMessage of that exception contains the url to open a new tab to. The tab opens to mitro-core/TwoFactorAuth with a login token signed by “signingKey” from the servlet TwoFactorAuth. The user inputs a verification or backup code, and if it is correct is to the login page with the same token, except the twoFactorAuthVerified part of the token has been changed to true, from false. The login page parses for a token and token_signature in the url, and if it’s there (which means 2fa was just completed), it checks the signature and token. If the password inputted the first time the login screen was displayed is correct, the user is logged in.
When 2fa is enabled and a user changes passwords, the user must first authenticate with 2fa. It works the exact same way as for logging in, except for the way the user is redirected. A request is made to the server, asking whether 2fa is enabled. If yes, a response is sent back to the extension containing a url to open a tab to (vs throwing an exception and getting the url from the exception message).

Questions

- Should we build this on top of an existing protocol? E.g. is this just a pretty interface to PGP? X509? SSH keys?
- Do we identify users by email address? If so, we need to verify the email address in some way (e.g. Google Account, sending them an email). What if they have multiple addresses? What if they change it?
- For the public service: Do people own their own account (GitHub) and belong to (multiple?) organization(s)? Do organizations own their accounts and the client can sign in to multiple accounts (Google)?
- If there are multiple servers, and the client is signed in to more than one, how do we know what user@company.com means? What if someone creates that account both on our public server, and on a private server?
- How do we authenticate users? Some enterprises may want different policies.
- How do we deal with unique ids (e.g. password ids)? Should we be worried about leaking data that way? Ideally, ids would be scoped per-user, but I don’t see how to make that work with sharing (easily).
- Can we use two-factor authentication in a smart way?
- How should we allow people to recover keys in case they forget their password? How should we permit organizations/enterprises to recover keys?
- How should the extension verify the site before it fills the form and clicks submit? Example: http://blog.chromium.org/2008/12/security-in-depth-password-manager.html

- Should we require requests to be signed with the private key? This proves the user has access to it. Should we also require another proof of identity? E.g. would could require BOTH a Google Account login and the key? Does this improve security?