DisOrder Cookie-Based Web Login Design

Introduction

This is the design for DisOrder's switch to a cookie-based login system.

We don't like HTTP basic authentication because:

- the browser UI as specified and implemented is awful
  - you get to display one string to the user
  - you can't log out
- authentication is mandatory: there is no facility for a "guest" user with reduced capabilities
- you need a separate `htpasswd` file

Cookie Protocol

Rather than store any session state in the CGI or server we keep it all in a signed cookie. Keys are stored in the DisOrder server (with its private UID/GID) and are not available to the CGI (which may well share a UID with other CGI programs).

Signed Cookies

The signing is done by the server, *not* the CGI: once the CGI has logged in as a user (in the normal way) it retrieves a cookie from the server and on subsequent requests offers that in place of the normal login protocol.

Let:

- K be the long-term signing key, stored only in the server
- T be the expiry timestamp, in hex. This will be a configurable amount of time in the future.
- U be the username (which must not have a "$" in it)
- P be U's password

Then:

- B = T + "$" + U + "$"
- g = base64(HMACSHA1(K, B + P))
- c = b + g

So the resulting cookie will look like `timestamp$username$signature`.

To verify a cookie c':

- Check that c' has not been revoked
- Split c' into T'U'g'
- Check that T' is not in the past
- Check that U' is a valid user and find their password P'
- Let B' = T' + "$" + U' + "$"
- Let g'' = base64(HMACSHA1(K, B' + P'))
- Verify that g' = g''

Notes:

- changing your password invalidates all existing cookies (deliberately)
- cookie values must be quoted in HTTP with the current choice of separator
- logouts are enforced by keeping a revocation list of cookies
- the only parsing done is splitting on semicolons and converting the timestamp from hex
- our base64 encoding is chosen to avoid HTTP separator characters, so the cookie need not be quoted

MDW designed the cryptography in this scheme for me.

Key Rollover

The signing key will expire after a configurable interval and a new one generated. In order to keep rollover transparent to users, old signatures will still work up to a point. We implement this simply by keeping the most recently expired key and checking with that if the verification with the current key fails. If the login lifetime is shorter than the key lifetime then this won't work.

New keys are also generated when the server is restarted. No provision is made for the old cookies remaining usable.

There is no provision for cryptographic or protocol agility; any change would need a server restart, resulting in all existing cookies being invalidated anyway. The CGI just sees the cookie as an opaque string.

User Management

New `users` Database

Currently passwords are regarded as configuration rather than state. This will change.

There will be a new database, `users`. This will be a `DB_HASH`. The keys will be usernames and the values will be the usual URL-encoded key-value pairs.

The following keys are defined:

Key	Meaning
password	This is the user's password. If this is not set the user has no password and can always log in.
rights	This is a comma-separated list of rights the user has. If this is not set the user has no rights at all.
email	The user's email address. This doesn't have to be set.
confirmation	The most recent confirmation string.
created	The timestamp that this user was created.

Undefined keys are never allowed; since some keys may have special properties, it's not appropriate to allow arbitrary keys to be used.

The `users` database is created mode 0600. When we join the database environment also we specify a mode of 0600, so that log files etc remain private to the server. Also on startup all files in the database home that except sockets have group and world permission bits cleared.

If a user has a `confirmation` key then they can **not** log in even if they have the right password.

`disorder-dump` will dump and restore the contents of `users`, and will make the dump file have mode 0600, since it now contains passwords.

User Rights

The following user rights are defined:

Right	Meaning
read	Allows all read-only operations.
play	Allows tracks to be played (added to the queue)
move any	Allows tracks to be moved in the queue.
move mine	Allows tracks to be moved in the queue if you submitted them.
move random	Allows tracks to be moved in the queue if they were randomly chosen.
remove any	Allows tracks to be removed from the queue.
remove mine	Allows tracks to be removed from the queue if you submitted them.
remove random	Allows tracks to be removed from the queue if they were randomly chosen.
scratch any	Allows the playing track to be scratched.
scratch mine	Allows the playing track to be scratched if you submitted it.
scratch random	Allows the playing track to be scratch if it was randomly chosen.
volume	Allows the volume to be changed.
admin	Allows administrative commands such as `reconfigure` and `shutdown`
rescan	Allows the `rescan` command.
register	Allows the `register` command.
userinfo	Allows fetching or editing of own user information.
prefs	Allows editing of track preferences
global prefs	Allows editing of global preferences

In order to implement this the command table will be extended with a column detailing which right(s) allow each command. No command needs more than one right to run; some commands (e.g. `scratch`) will allow one of several possible rights and do more specific checking themselves.

Additionally the special right `all` allows users who have it to do anything. The automatically created `root` user has this right.

Undefined rights are not allowed (making downgrade across a change to the set of rights problematic).

Users without `read` aren't specifically prohibited but we don't guarantee that they will be able to to anything useful.

Rights are implemented as as property of connections. This has implications for commands that change them.

Default Rights

Default rights for new users are configurable. If there is no override then they are all rights except `admin` and `register`, with the move/remove/scratch rights determined as follows:

- if `restrict scratch` then `scratch mine` and `scratch random`; otherwise `scratch-any`
- if `restrict remove` then `remove mine`; otherwise `remove any`
- if `restrict move` then no `move *` at all; otherwise `move any`

This logic is supposed to preserve pre-existing configurations. `restrict` will generate an error message but not inhibit startup. In a distant future version it will be removed.

Upgrading

If the `users` database does not exist on startup then it is constructed from the `allow`, `trust` and `restrict` settings.

The default rights for users from `allow` settings are computed as follows:

- The `www-data` user is ignored (with a log message)
- `root` always gets `all`
- If a user is listed in `trust` then they get `all`
- Otherwise you get default rights as described above

If `allow` or `trust` are found then the will provoke an error message but not inhibit startup. In a distant future version they will be removed completely.

Special Users

If `root` does not exist then it is created at startup with a random password and rights set to `all`. The local (UNIX) root user can extract the password directly from the database, so it need not be written to any configuration file.

`guest` need not exist and is not created automatically, but is treated specially by the web interface; see below. A common configuration would be to create it with no password and with rights of `read` or `read` and `register`.

`disorder setup-guest` (or something) will painlessly create a guest user, an option determining whether registration is allowed or not.

New Commands

The `become` command will be removed.

adduser

`adduser USERNAME PASSWORD`

This command creates a new user. If the password is the null string then no password is set. Default rights are set from the configuration file. Usernames containing semicolons and perhaps other characters will be prohibited.

This command requires the `admin` right and only works on local connections.

`disorder adduser` will call this command and `edituser` if an email address is supplied.

If the user exists the command fails even if they are an unconfirmed user (see below).

On success the response is 250.

deluser

`deluser USERNAME`

This command deletes a user.

This command requires the `admin` right and only works on local connections.

It **is** possible to delete unconfirmed logins.

`disorder deluser` will call this command.

When a user is deleted, all connections for that user have their rights set to 0, making them useless. The client should only assume this has been done once it sees the response.

On success the response is 250.

edituser

`edituser USERNAME KEY VALUE`

This command edits some property of a user.

A user with the `admin` right may use this command with any `USERNAME` and `KEY`. A user with the `userinfo` right may use this command with their own `USERNAME` and `KEY` of either `password` or `email`. No other use is allowed.

`disorder password` and `disorder email` will call this command.

When user has their rights changed, all connections for that user have their rights modified accordingly, but the connections are not destroyed. The client should only assume this has been done once it sees the response.

When a used has their password changed, all connections for that user are immediately disconnected. Again, the client should only assume this has been done once it sees the response.

On success the response is 250.

userinfo

`userinfo USERNAME KEY`

This command retrieves some property of a user.

A user with the `admin` right may use this command with any `USERNAME` and `KEY`. A user with the `userinfo` right may use this command with their own `USERNAME` and `KEY` of either `rights` or `email`. No other use is allowed.

`disorder userinfo` will call this command.

On success the response is "252 VALUE". If the key is not found the response is 555 (as for e.g. `get` and `get-global`.)

register

```
register USERNAME PASSWORD EMAIL
```

This command creates a new user.

This command only works on local connections and requires the `register` right. The user is created with a random `confirmation` value and won't be able to log in yet. Probably the `confirmation` value includes the username to allow it to be efficiently found.

If the user already exists, but still has a `confirmation` value, and `created` indicates that the registration is some configurable age or older, it is overwritten.

It is possible to configure an explicit list of acceptable email domains, but by default any email address is allowed.

The response is the confirmation string. The server will send an email to the email address asking the user to visit a URL including this string to activate their login. The confirmation string is `base64(USERNAME;RANDOM)` (but this a private arrangement between `register` and `confirm` and isn't guaranteed).

There will be no command-line equivalent of this command.

On success the response is "252 CONFIRMATION"

confirm

```
confirm CONFIRMATION
```

This command activates a login created with `register`. It can be used without logging in. The `confirmation` key is removed from the relevant user and the user is then logged in and in particular may use `cookie`.

There will be no command-line equivalent of this command.

On success the response is "232 USERNAME".

reminder

```
reminder USER
```

This command makes the server (not the CGI) send a password reminder email to the user's registered email address. It only works if they have an email address and a nontrivial password. Reminder emails are rate-limited per user.

On success the response is 250.

make-cookie

```
make-cookie
```

This command generates a cookie for the logged-in user as described above. It can be used later in the `cookie`

command to log in.

On success the response is "252 COOKIE".

cookie

```
cookie COOKIE
```

This command verifies the cookie as described above and if it passes, the user is logged in.

On success the response is "232 USERNAME", so that the caller knows who they logged in as (so the web interface doesn't have to know the format of the cookie).

revoke

```
revoke
```

Stores the cookie used to login in an internal revocation list, until it expires. The cookie will not be usable to login. Only works if login was via `cookie` rather than `user`.

On success the response is 250.

users

```
users
```

Lists all known users in no particular order.

Web Interface

The web interface can now be in one of two modes. These mostly concern logging in and registration: the enforcement of who-can-do-what policy is implemented by the server.

Guest Operation

If no cookie arrives then the web interface operates in guest mode. It logs into the server as `guest` with the empty string as password (i.e. not using the cookie commands). A new menu item **Login** appears at the top of the screen.

The **Login** page contains a username/password form and a link to a **Registration** page. It uses the `make-cookie` command to log in and sends back the cookie to the client. If the `make-cookie` fails it returns to the **Login** page but additionally displays an error message. On success it goes back to the front page by default, or to the `redirect` URL.

The **Registration** page can be suppressed by not giving the `guest` user the `register` right. It contains a form asking for username, password and email address. If the `register` command fails it returns to the **Registration** page and additionally displays an error message. On success it displays a confirmation page explaining the email check.

Logged-In Operation

If a cookie does arrive then the CGI logs into the server with the cookie. On error it goes straight to the **Login** page, setting a `redirect` argument so the user ends up back in the right place when they log in.

There is an extra **Account** page. This contains a form for changing email address and/or password and a logout

button.