Security

Contents

Restricting Route Access

Please refer to the Routing section for the instructions on securing the routes.

Password Hashing

Password hashing and comparison is handled by the `buddy.hashers` namespace provided by the Buddy.

It provides two functions, `encrypt` and `check`. The first will encrypt and salt the password, while the second compares the raw password to the encrypted string generated by the first. BCrypt is used as the default encryption algorithm.

```
(ns myapp.auth
  (:require
   ...
   [buddy.hashers :as hashers]))

(def raw "secret")
(def encrypted (hashers/derive raw))
(hashers/check raw encrypted)
```

The `encrypt` function allows specifying additional parameters such as the algorithm and the number of iterations:

```
(hashers/derive "secretpassword" {:alg :pbkdf2+sha256})
(hashers/derive "secretpassword" {:alg :pbkdf2+sha256
                                  :salt "123456"})
(hashers/derive "secretpassword" {:alg :pbkdf2+sha256
```

Build Tool: lein ▾

Topics

Your First Application
REPL Driven Developme
Application Profiles
HTML Templating
Static Assets
ClojureScript
Routing
RESTful Services
Request types
Response types
Websockets
Middleware
Sessions and Cookies
Input Validation
- **Security**
Component Lifecycle
Database Access
Database Migrations
Logging
Internationalization
Testing
Server Tuning
Environment Variables
Deployment
Useful Libraries
Sample Applications
Upgrading
Clojure Resources

Books

```
                                             :salt "123456"
                                             :iterations 200000})
```

The following algorithms with their associated options and defaults are seen below:

`:alg :bcrypt+sha512`

`:iterations` 12

`:salt` random

`:alg :pbkdf2+sha256`

`:iterations` 100000

`:salt` random

`:alg :pbkdf2+sha3_256`

`:iterations` 100000

`:salt` random

`:alg :pbkdf2+sha1`

`:iterations` 100000

`:salt` random

`:alg :scrypt`

`:salt` random

`:cpucost` 65536

`:memcost` 8

`:parallelism` 1

`:alg :sha256`

`:salt` random

`:alg :md5`

`:salt` random

For information on restricting access to specific routes, please refer to the routing section.

For an alternative security solution you may wish to check out the Friend library.

LDAP Authentication

The following example demonstrates how to authenticate with the `sAMAccountName` using the clj-ldap library.

First, we'll add the following dependency to your `project.clj`.

```
 [org.clojars.pntblnk/clj-ldap "0.0.12"]
```

Next, we'll need to require the LDAP client in the authentication namespace and `defstate` to hold the connection pool:

```
(ns ldap-auth
  (:require
    [mount.core :refer [defstate]]
    [clj-ldap.client :as client]))
```

We'll define our LDAP host as follows, note that the `host` key points to a vector of LDAP servers.

```
(def host
  {:host
   {:address         "my-ldap-server.ca"
    :port            389
    :connect-timeout (* 1000 5)
    :timeout         (* 1000 30)}})
```

We'll can now declare a connection pool for LDAP:

```
(defstate ldap-pool :start (client/connect host))
```

Finally, we'll write a function to authenticate the user using the above declared pool.

```
(defn authenticate [username password & [attributes]]
  (let [conn           (client/get-connection ldap-pool)
        qualified-name (str username "@" (-> host :host :address))]
    (try
      (if (client/bind? conn qualified-name password)
        (first (client/search conn
                              "ou=people,dc=example,dc=com"
                              {:filter     (str "sAMAccountName=" user
                               :attributes (or attributes [])})))
      (finally (client/release-connection ldap-pool conn)))))
```

The `attributes` vector can be used to filter the keys that are returned, an empty vector will return all the keys associated with the account.

Cross Site Request Forgery Protection

CSRF attack involves a third party performing an action on your site using the credentials of a logged-in user. This can commonly occur when your site contains a malicious link, a form button, or some JavaScript.

Ring-Anti-Forgery is used to protect against CSRF attacks. Anti-forgery protection is enabled by default.

Once the CSRF middleware is enabled a randomly-generated string will be assigned to the *anti-forgery-token* var. Any POST requests coming to the server will have to contain a parameter called `__anti-forgery-token` with this token.

The `<app>.layout` namespace of your application creates a `csrf-field` tag that can be used to provide the token on the page:

```
(parser/add-tag! :csrf-field (fn [_ _] (anti-forgery-field)))
```

We can use it in our templates as follows:

```
<form name="input" action="/login" method="POST">
  {% csrf-field %}
  Username: <input type="text" name="user">
  Password: <input type="password" name="pass">
  <input type="submit" value="Submit">
</form>
```

Any requests that aren't `GET` or `HEAD` and do not contain the token will be rejected by the middleware. The server will respond with a 403 error saying "Invalid anti-forgery token".

The anti-forgery middleware is wrapped around the `home-routes` in the `app` definition of the `<app>.handler` namespace. Note that the `wrap-csrf` wrapper will be applied to `home-routes` explicitly, and should be applied to any other route groups that are not meant to be used by external clients.

```
(def app
  (-> (routes
        (wrap-routes home-routes middleware/wrap-csrf) ;;wraps CSRF pr
        base-routes)
      middleware/wrap-base))
```

If you wish to disable it for any reason then simply update the `app` definition as follows:

```
(def app
  (-> (routes
        home-routes ;;no CSRF protection
        base-routes)
      middleware/wrap-base))
```

Please see here on details how to enable CSRF for select routes in your application.