# Simple chat app

The application can be found from <https://github.com/pcmakine/cybersecuritybase-project>. It is a very simple one. It allows a user to signup (<http://localhost:9090/signup>), login (<http://localhost:9090/login>), view his or her profile ([http://localhost:9090/profile/<id](http://localhost:9090/profile/%3cid)>) and view messages that she or other users have written (<http://localhost:9090/chat>). When the application starts two test users and two messages are created. The first user has username alice and password alice. The second user has username bob and password bob.

As instructed the project has five flaws from the owasp top ten list. Most of them I was not able to identify with the owasp ZAP, but below you will find steps on how to reproduce each one, and how they could be fixed.

**A1-Injection:** There is currently an sql injection vulnerability in the signup page. The attacker can, for example, drop a table if she knows or can guess a table’s name in the application.

Steps to reproduce:

1. Navigate to <http://localhost:9090/signup> and fill in a username (something else than alice or bob as they are taken already) and password. The email field is the one where we can inject sql
2. Fill into the email field for example a'; DROP TABLE ACCOUNT --
3. You will get an error page because the account table has been dropped and the application is trying to insert the new user into it
4. You can restart the application to recreate all the tables

How to fix:

This error has been introduced into the application probably because a new developer unfamiliar with spring had to implement a check for the uniqueness of the email address when a new user signs up. The check has been implemented with a raw sql query instead of a prepared statement or using the JpaRepository methods as in checking for the uniqueness of the username. One solution would be to add in Account repository a method with the following signature: findByEmail(String email). This could be used in the signupController’s emailExists method in the same way as the findByUsername method is currently used in the submitForm method. Even better solution might be to do both with a single sql query with a prepared statement.

**A4-Insecure Direct Object References and A7-Missing Function Level Access Control**

Currently the profiles are meant to be seen only by the users themselves, but there is a flaw allowing users to see other users’ profiles as well.

Steps to reproduce:

1. Navigate to <http://localhost:9090/login>
2. Input alice in the username field
3. Input alice in the password field
4. You will be redirected to <http://localhost:9090/chat>
5. Click the link “My profile” at the top
6. You will be redirected to <http://localhost:9090/profile/1>
7. Try to change the url to <http://localhost:9090/profile/2>
8. You will see all Bob’s information (including email which should be kept private), even though you are logged in with Alice’s credentials

How to fix:

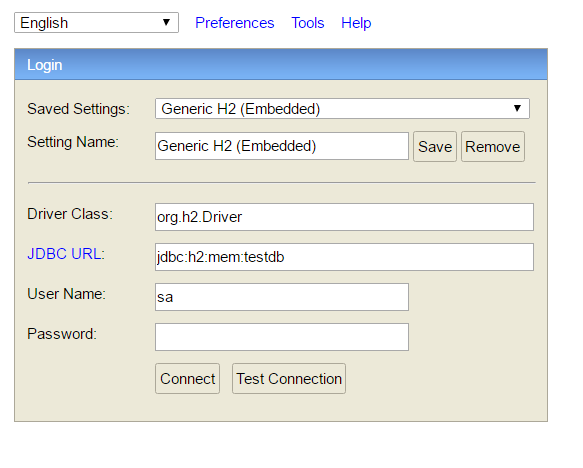
When the user tries to access the url ending profile/<id>, the server should verify that the user trying to access the resource is indeed the one with the given id. If not some kind of an error message should be returned to the user.

**A5-Security Misconfiguration**

At the moment, the application’s security is badly configured. Any user can access the database through the web interface.

Steps to reproduce:

1. Login for example with alice’s credentials
2. Go to url <http://localhost:9090/h2-console/>
3. Input the fields as in the below screenshot



1. Click connect
2. You have full access to the database (it’s an in-memory db, so everything will be reset when you restart the app)

How to fix:

In the SecurityConfiguration class in method configure, we should delete the rows

http.csrf().disable();  
http.headers().frameOptions().sameOrigin();

and

.antMatchers(**"/h2-console/\*"**).permitAll()

Then the users could not access the database anymore, plus we would have the csrf protection as well.

**A6-Sensitive Data Exposure**

The application does not use encrypted connection, so for example the passwords can easily be intercepted in plain text.

Steps to reproduce:

This can be verified by setting the browser to use localhost port 8080 as the proxy (if you have for example the owasp zap running in that port in your localhost) and logging into the application. Now you can see the request including the password in the owasp zap user interface.

How to fix:

The solution would be to use https.

Additionally, related to A6, the auto complete feature is not disabled from the password fields. This is actually a flaw that owasp Zap finds too.