Security report

Security breach	Covered?
Protection against malicious file uploads	Yes
Protection against Man-in-the-middle attacks	Yes
Protection against Link Injection Protection	Yes
Protection against Attribute autocomplete	Yes
Protection against SQL Injection	Yes
Authentication and protections against unauthorized API calls	Yes
Session hijacking protection	Yes

Protection against malicious file uploads

To protect users against malicious file uploads, we sanitize all the file uploads in FileResource.java by creating a whitelist of only the allowed extensions that can be relevant for the Service provider and the customer, Ex:.pdf, .png => No .exe or .dmg files are uploaded. For extra precaution, we also check it in the front-end and it'll give an error message if user are uploading wrong file types.

Protection against Man-in-the-middle attacks

For the Man-in-the-middle attacks we find the best way to prevent it by make oor deployment server (Previder) use only HTTPS(this way all traffic is encrypted and protected against replay attacks) because with HTTP an attacker can still intercept requests to send information

Protection against Link Injection

In order to resolve link injection issues, we have limited the accessibility of user input and its control over the internal state of the server or database by only using and accepting a limited set of HTTP requests GET,POST,PUT and DELETE. Since we are paranoid over security, we managed to pair all the requests that a customer (a.k.a user) makes with the above mentioned methods so that there is no room for malicious links or embedded links like CSRF (Cross-Site Request Forgery)

Protection against Attribute autocomplete

In our application the only possible place for attribute autocomplete attacks is in the Application form but we've already turned off autocomplete information. And most of user's sensitive informations are required to upload in form of additional document The attack against the autocomplete happens through accessing the cache, and this only happens if the attacker can access the victim's machine, which in our case is not quite possible since we are limiting the sensitive data to be visible for not-logged in users.

Protection against SQL Injection

In our web application we ensure that 100% of all our SQL codes are using Prepared statements in order to not mix the input from the user with code. This is used extensively inside of all the Data-Access-Object files that contain methods to access and update the database. Also, we are sanitizing user input by specifying the type of the input and also the mediatypes of the used media and Java constructor, get, set method also help the sanitization.
The combination of prepared statements and the user input sanitization we have in our application makes an SQL injection quite unlikely and makes the web application secure and functional.

Authentication and protect against unauthorized api calls

An attacker might not use the website to make calls to the server at all and create his own requests to the backend. Thus we need to check in the database the callee is authenticated to have access to getting or changing a specific resource.

To Authenticate users we have them login using their email and password, the password is hashed using sha512 with salt stored for each user. To maintain the authentication of users while they're still alive in the database we store a securely random authentication token in the database that is meaningless to the content of the password and in the user's cookies so we can later verify the user is still logged in. These tokens expire after a while to limit the risk of them being found and used by someone who did not login.

When an api call is made to access a resource that not just anyone should be able to access, the authentication token is used to get the user_id of the currently logged in user and a query to the database is done to check whether this user has access to the resource. Due to lack of time we weren't able to do this for all resources that needed it. But to give an example we did add these checks for accessing the userinfo(email, name and surname). (See classes dao.userDao and unit.dao.TestUserDao)

Session hijacking protection

To protect against this attack we have 2 steps protection:

+ Sanitize all user input displayed on the site to prevent xss. (mention in SQL injection protection)
+ Unique, meaningless, unpredictable authentication session token (generated by Java SecureRandom with randomToken() method in AuthenticationDao) makes sure that when a user isn't logged in their data can't be accessed (check in login() in UserController.java). Because we are extra cautious we also prevent any viewing of our site from within iframes.