Technische Universität München
Fakultät für Informatik

Exercise Sheet 7: Software Security

Welcome to the seventh week of Advanced Topics of Software Engineering. This week, we will focus on topics of information security. For this matter, we introduce a fictional scenario, where a vulnerable file sharing website, *MosFilesley*[1], is going to be hacked by you! For the practical exercise, you will see how to send a JWT using HTTP Only Cookies, protect your endpoints from Cross Site Request Forgery (CSRF) attacks, and configure Cross Origin Resource Sharing (CORS) such that your back-end only serve requests from a trusted front-end source.

To demonstrate how to model the threats imposed by your attacks, we further discuss *attack trees* and the *Common Vulnerability Scoring System (CVSS)* which scores vulnerabilities by their severity.

Theoretical Exercise

Exercise 1: Attack Trees – Modeling Threats

The owners of the popular, but controversial, file sharing website *MosFilesley* noticed that their website is under constant cyber attacks from unknown perpetrators. “Since last Monday we are being hacked on a daily basis, why would anyone do that? We are a legitimate business trying to turn in a profit.”, says an anonymous source in the company.

To be able to prevent these attacks, it is important to get an understanding of the vulnerabilities of the website. One possible way to model potential threats are *Attack Trees*, which you were introduced to in the lecture. These can be used to outline and evaluate weaknesses of the system and guide improvements in the implementation.

You have been hired to improve MosFilesley’s defenses. In the following exercises you will (i) identify attack goals, (ii) create a threat model using attack trees, and (iii) investigate concrete instances of possible attacks. Figure 1 shows the user interface (UI) of the website.

1. After the first phone call you only know there are user accounts, a restricted area, and files stored somewhere in MosFilesley. For you this is already enough information to identify at least three possible attack goals.
2. Try to think of all attacks against one of these goals and build the respective attack tree. Each goal is the root node of an attack tree. Trees can share nodes and subtrees.
3. What could be possible attacks against authentication or user accounts that are independent of the implementation of the Web application itself?

   Note: Also, think of the infrastructure where the Web application is deployed and how it is being served.
4. The *Common Vulnerability Scoring System (CVSS)* provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting its severity with respect to ease of exploit and the impact of exploit[2]. It uses different properties (e.g. can the attack be exploited over the

[1] MosFilesley: https://mos-filesley.sse.in.tum.de

[2] If you want to know more, https://en.wikipedia.org/wiki/CVSS has a good example. The official guide can be found here: https://www.first.org/cvss

Mos Filesley

Hello root (Logout)!

What file do you want to share today?

Active accounts

Leia writes: "Nice Pic!"

[Download file]

Luke writes: "Hawt Sauce!"

[Download file]

Figure 1: The only picture of MosFilesley.

network, does the attacker need a valid user account on the system or how much information can the attacker access) to rate vulnerabilities. Scores range from 0 to 10, with 10 being the most severe.

In the following, you will use the CVSS Calculator at https://www.first.org/cvss/calculator/3.1 to rate **one** of your identified attacks. For this you must specify the values of the following basic metrics that are necessary to calculate the *Base Score*:

- **Ease of Exploit:** Attack Vector, Attack Complexity, Privileges Required, User Interaction
- **Impact of Exploit:** Scope, Confidentiality, Integrity, Availability

Briefly discuss why you chose a specific level for each metric.

Exercise 2: Web Security Vulnerabilities

After your profound investigation of MosFilesley (https://ase.sebis.in.tum.de), you are confident to achieve at least the following four goals:

1. Bypass the Authentication and gain access to MosFilesley.
2. Obtain a copy of the MosFilesley member list (Hint: Remember on Unix, *Everything is a file*).
3. Remove all files stored on MosFilesley.
4. Find a way to log the username and session IDs of everyone who accesses the website. Hint: In order to avoid problems with the Same-Origin-Policy and Mixed-Content you can use https://ase.sebis.in.tum.de/logger.php to dump information.

Further Notes

- Please do not DoS the system.

- If the main website was hacked, you can use http://ase.sebis.in.tum.de:999[0-7]/ as a fall back. They are all identical copies.
- The system will restore itself every 30 minutes (every full and half hour). If you accidentally break the system, write an email to burak.oez@tum.de.

Practical Exercise

Exercise 3: Including Java Web Token (JWT) in HttpOnly Cookie

From the last exercise, you have seen how JWT, particularly Java Web Signature, is created in the backend. Since JWT is used to authenticate or authorize requests from clients, we need to securely store the JWT in the client browser or system. In the browser, keeping JWT in localStorage or sessionStorage implies several security risks, like the JWT can be read and manipulated using Javascript. Another option is to store the JWT as a redux state, which keeps the JWT in the memory. However, the memory does not persist data, so if users refresh the page, the JWT will be erased. As a result, our client app has to request the JWT from the server again.

In contrast to the above techniques, HttpOnly Cookies[3] intends to store data on the client without allowing the client script to read and modify the token. Note: different web browsers can handle HttpOnly Cookies differently. For instance, some browsers still let the client reads HttpOnly Cookies. HttpOnly Cookies are created by the server and attached to the response of a request. When the client browser receives the response, it keeps the HttpOnly Cookies and includes them in every future request sent to the server. Using HttpOnly Cookies, the browser persists the data when a user refreshes a webpage while keeping the data away from third parties if the browser handles HttpOnly Cookies correctly.

In this exercise, we will explore how to:

- Insert the generated JWT token into an HttpOnly Cookie in Spring
- Filter the JWT Token and extract the user's identity from incoming requests

Create an HttpOnly Cookie containing a JWT

In the authentication service (i.e. in the processing of the `/auth` endpoint), after generating the JWT, create a new Cookie using the following command:

```
1    Cookie jwtCookie = new Cookie("jwt", jwt);
2    // TODO: Configure the cookie to be HttpOnly and expires after a period
3    // Then include the cookie into the response
```

The first parameter of the Cookie constructor is the cookie name, and the second parameter is the cookie value.

You can learn more about how to configure the properties of cookies in the following tutorial[4]. Can you set the `jwtCookie` to become an HttpOnly cookie? And then add the cookie into the response?

Send a request to the `/auth` endpoint with a valid user credential using Postman. If the HttpOnly Cookie is configured properly, you will see an output similar to Figure.

When a client receives our HttpOnly Cookie, it will attach the cookie in every upcoming request. Therefore, we need to read this JWT value in the cookie to process it. You can read the cookie from incoming requests in the Filter from Exercise 6, or use other approaches. To read the cookie from a HttpServletRequest in Spring, use the following command:

```
1    Cookie[] cookies = request.getCookies();
```

[3] HttpOnly Cookies: https://owasp.org/www-community/HttpOnly
[4] Cookie in Spring Boot: https://dzone.com/articles/how-to-use-cookies-in-spring-boot

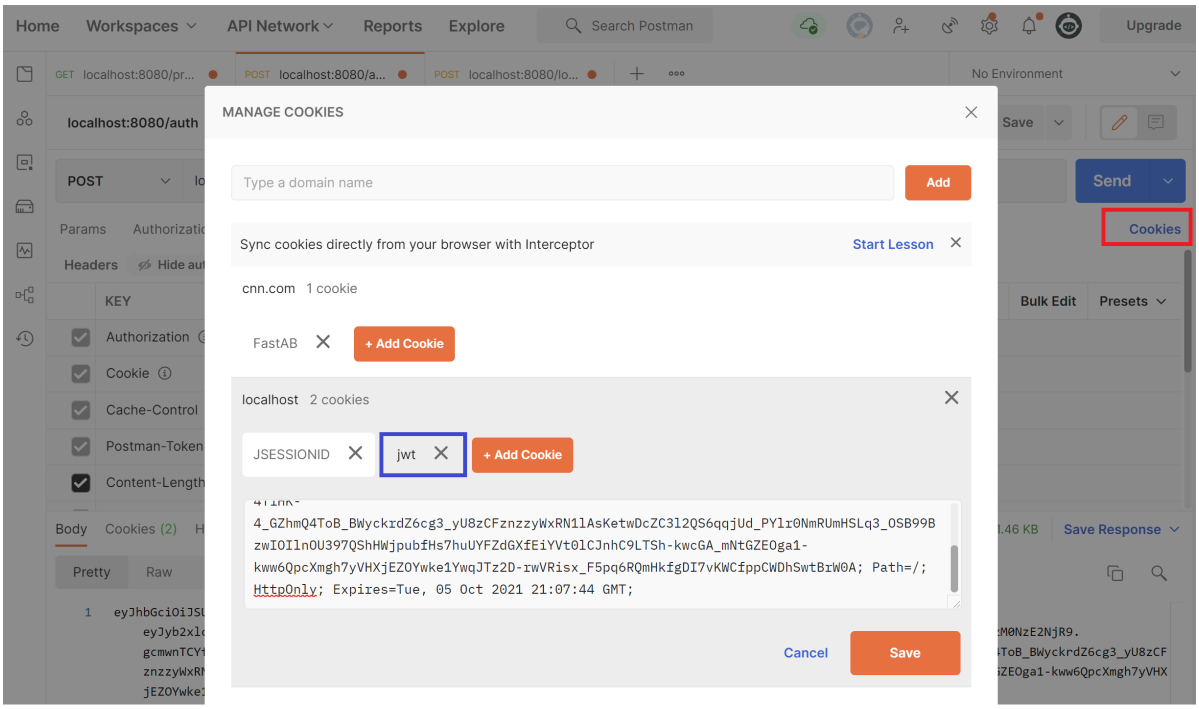

Figure 2: An HttpOnly Cookie is displayed in Postman. Here you can see the JWT token, HttpOnly tag, and other properties of the cookie.

Using the JWT from the HttpOnly Cookie, your application should authenticate the user successfully like using the JWT from an Authorization header in Exercise 6. Try to send requests to other project endpoints and check if you receive the expected output.

Exercise 4: Enable Cross-Site Request Forgery (CSRF) Protection in Spring

In the last exercise, we have disabled CSRF in Spring Security to focus on the authentication and authorization using JWT. Now we will enable and implement CSRF protection to secure our application.

Spring Security's default Cross-Site Request Forgery (CSRF) protection which requires an additional CSRF token on non-idempotent HTTP methods (e.g. POST, PUT) to implement the *Synchronizer Token Pattern*[5].

At the moment, creating a new project should work fine as CSRF protection is deactivated. As this is bad and insecure, it is common practice that servers randomly generate CSRF tokens when providing HTML forms. The token is then submitted together with the form submission (e.g. HTTP POST) and the server checks for its validity. A form provided by an attacker will not be rendered by the server and thus not contain a valid CSRF token.

This does not work for Single Page Applications (SPA) (e.g. Angular), as the client is responsible for rendering forms. A common anti-CSRF technique for these JavaScript clients is illustrated by the high-level process in Fig. 3. The basic idea is that the server sends a randomly generated authentication token in a cookie to the client, who adds a custom request header with the token in all subsequent requests (not required for safe/idempotent methods, e.g. GET). This technique is effective because all browsers implement the same origin policy. Only code from the website on which cookies are set can read the cookies from that site and set custom headers on requests to that site[6].

The implementation of the outlined process is easier than the theoretical concept. First, let us enable CSRF protection with the CSRF token included inside a cookie in the `SecurityConfig`:

```
1 @Configuration
2 public class SecurityConfig extends WebSecurityConfigurerAdapter {
3
4     @Override
```

[5]CSRF Protection and Synchronizer Token Pattern: https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#csrf-explained
[6]CSRF Protection for JavaScript Clients: https://angular.io/guide/security#xsrf

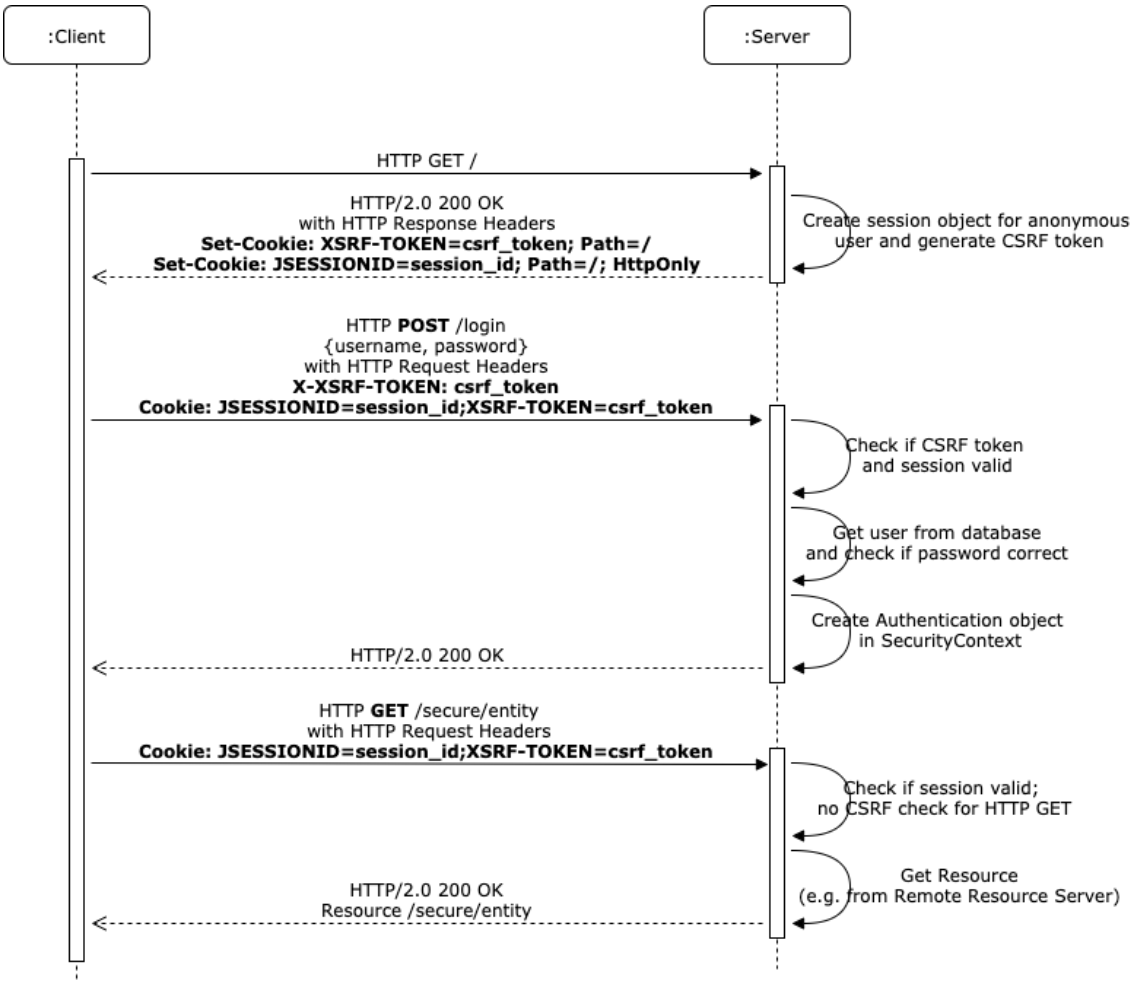

Figure 3: Authentication with CSRF Protection for Form Login on JavaScript Clients

```
    protected void configure(HttpSecurity http) throws Exception {
    super.configure(http); // keep defaults, but partly override below
    http
        .csrf()
          .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
        // Other configurations
    }
}
```

Afterward, you should not be able to log in nor create a new project. Do not worry, that is how it should be because CSRF token is required when making HTTP requests that modify the resources, i.e., POST, PUT, UPDATE, DELETE, or PATCH requests. However, requests that do not change resources like GET or OPTIONS do not require CSRF tokens.

Although the authentication request is unsuccessful, there is an XSRF-TOKEN cookie with a random String value in the request header (you can also see this in Postman Cookies). Create a new Header named X-XSRF-TOKEN, which is the default header name for Spring Security to recognize CSRF tokens from the client. Paste the random as the value of the X-XSRF-TOKEN, send the request to the backend, and your login request should be successful. That is because you have specified as the legitimate client of the Spring application with its CSRF token in the request header.

Obviously, this is not a standard approach to use CSRF tokens for securing the authentication and create project endpoints in your application. The common procedure is:

1. Create a GET request for initializing the CSRF token.
2. The client calls the GET request to obtain the CSRF token
3. The client creates an HTTP Header with a CSRF Header name recognized by the server (X-XSRF-TOKEN in Spring by default), then set the CSRF token in step 2 as the value
4. The client sends the request to the backend

To finish this exercise, implement the above procedure to obtain a CSRF token and call the authentication and project creation services.

Exercise 5: Secure Application with Cross-Origin Resource Sharing (CORS) Protection

CORS is a header-based mechanism to help server allows browsers from specified origins to load the server's resources. An origin is indicated by its domain, scheme or port, e.g., `https://localhost:8080` origin has `https` as the scheme, `locahost` as the domain and `8080` as the port. CORS helps us to specify that only our front-end our trusted web applications are allowed to access the services from the server. You can learn more about CORS at https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

An interesting access control scenario in CORS is sending requests with credentials[7]. This type of request allows cookies in the header, thus we can send the cookies back to the server. To craft credentialed requests, the server needs to specify the trusted origin in a `Access-Control-Allow-Origin` header. In other words, stating `Access-Control-Allow-Origin: *` **will NOT enable the credential request**. Secondly, the server must also set the header `Access-Control-Allow-Credentials` to **true** (Access-Control-Allow-Credentials: true). The outcome of this setting is, only trusted front-end applications are allowed to send requests with user credentials in Cookies (preferably HttpOnly).

Spring Security supports the configuration of CORS with the following properties. There are several ways to define CORS restrictions in Spring Application:

1. Controller Method CORS configuration[8]
2. Global CORS Configuration[9]

[7] Credentialed Requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#requests_with_credentials
[8] Controller Method CORS Configuration: https://spring.io/guides/gs/rest-service-cors/#controller-method-cors-configuration
[9] Global CORS Configuration: https://spring.io/guides/gs/rest-service-cors/#global-cors-configuration

Choose one of your favorite method to secure the endpoints of the Project application. After finishing your implementation, create a React Web application with the same host and port as the allowed origin of your configuration. If you are using **localhost** for your React application, since **localhost** is the same host as the backend but with a different port, you need to configure your fetch method such that it sends a request with **cors** mode and **origin-when-cross-origin** referrerPolicy. **origin-when-cross-origin** indicates that the client can have the same host as the server, which should not be allowed, but we specify this mode for demonstrating the use of CORS in this tutorial.

Next, send an authentication request to the server, and you should see the user is authenticated successfully. If you change the port number of the React application, the server shall not authenticate the user as the request does not come from a permitted origin.