**DHWorld OWASP Report**

Author: Antonio Denkov

Student №: 3995208

Table of Contents

**A1:2017 - Injection3**

**A2:2017 - Broken Authentication4**

**A3:2017 - Sensitive Data Exposure5**

**A4:2017 - XML External Entities (XXE)6**

**A5:2017 - Broken Access Control7**

**A6:2017 - Security Misconfiguration8**

**A7:2017 - Cross-Site Scripting (XSS)9**

**A8:2017 - Insecure Deserialization10**

**A9:2017 - Using Components with Known Vulnerabilities11**

**A10:2017 - Insufficient Logging & Monitoring12**

**References13**

**A1:2017 - Injection**

Injection is called an attacker’s attempt to send data to an application in a way that will change the meaning of commands being sent to an interpreter. For example, the most common example is SQL injection, where an attacker sends “101 OR 1=1” instead of just “101”. When included in a SQL query, this data changes the meaning to return ALL records instead of just one. There are lots of interpreters in the typical web environment, such as SQL, LDAP, Operating System, XPath, XQuery, Expression Language, and many more. Anything with a “command interface” that combines data into a command is susceptible.

To avoid this type of attack I use a ‘repository’ interface, which extends the Spring Boot provided JPA Repository interface. To enforce the SOLID principles and further restrict an attacker’s access, I also have a service interface. This interface is implemented in a service class, which uses the repository interface. This method ensures application security when a user is accessing their details, for example.

HQL is not used in the code and data is retrieved via mapping a whole object or a path variable.

**A2:2017 - Broken Authentication**

Broken authentication is an umbrella term for several vulnerabilities that attackers exploit to impersonate legitimate users online. Broadly, broken authentication refers to weaknesses in two areas: session management and credential management.

In my application I use Spring Security which makes use of HttpSecurity to provide mechanisms for efficient and secure authentication and session management

**A3:2017 - Sensitive Data Exposure**

Sensitive data exposure is a security threat occurs when the web application doesn’t adequately protect sensitive information like session tokens, passwords, banking information, location, health data, or any other similar crucial data whose leak can be critical for the user.

In the case of my web application, there are two user details which would require additional security: a user’s password and phone number. To provide sufficient password security I use the well known Bcrypt encoder. It uses a Key Factor (or Work Factor) which adjusts the cost of hashing, which is probably Bcrypt’s most notable feature. Bcrypt can expand the ‘Key Factor’ to compensate for increasingly more-powerful computers and effectively “slow down” its hashing speed. Changing the Key Factor also influences the hash output, so this makes Bcrypt extremely resistant to rainbow table-based attacks. Newer computers can attempt to guess the original input of the hash, but it would still take the same amount of time (or longer) to verify whether its guess is a match or not.

The user’s phone number is not encrypted at this point. To protect the phone number I can use Bcrypt to encrypt it the same way as the password.

**A4:2017 - XML External Entities (XXE)**

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In the case of my application, XML is very rarely used. Whenever possible, I use JSON for storing more sensitive information.

**A5:2017 - Broken Access Control**

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user.

In my application I use role-based access and store user’s information in a JSON Web Token (JWT). The way it works is as follows: The server generates a token that certifies the user’s identity, and sends it to the client. The client will send the token back to the server for every subsequent request, so the server knows the request comes from a particular identity.

**A6:2017 - Security Misconfiguration**

Security Misconfiguration is simply defined as failing to implement all the security controls for a server or web application, or implementing the security controls, but doing so with errors. What a company thought of as a safe environment actually has dangerous gaps or mistakes that leave the organization open to risk.

In the case of my application, to provide maximum security, I use role-based access and HttpSecurity configuration in order to let Spring Boot know when to authenticate all users, which filter to use, and which exception handler to be used.

**A7:2017 - Cross-Site Scripting (XSS)**

Cross-site Scripting (XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user’s browser.

I use React.js for my frontend, which provides decent protection against XSS attacks. To fight against XSS, React prevents render of any embedded value in JSX by escaping anything that is not explicitly written in the application. Prior to rendering, React.js converts everything to a string.

**A8:2017 - Insecure Deserialization**

Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.

It is even possible to replace a serialized object with an object of an entirely different class. Alarmingly, objects of any class that is available to the website will be deserialized and instantiated, regardless of which class was expected.

In order to protect the web application from this type of vulnerability, serialized object that can be manipulated by the user should never be passed to the Deserialize function. Instead of deserializing, a good alternative is to use a secure data interchange like JSON if it’s necessary to pass serialized data between the user and the web application.

Possible solutions for my case would be:

1. To include integrity checks: when possible, include positive validation based on signatures for serialized data.

2. To use a 3rd party protection tool. A good one is the Hdiv RASP Protection. Most of Insecure Deserialization attacks try to execute commands using input data that has been provided by the request or the database. Thanks to the data flow control provided by the tool, it is possible to understand how the data coming from the request or the database is used during the execution, thereby blocking any attempt to execute commands from such sources and totally avoiding this kind of issue.

3. Add a view model for each existing model

**A9:2017 - Using Components with Known Vulnerabilities**

Using components written by other programmers leads to certain risks of these components having security vulnerabilities. Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version.

Therefore, to avoid any possible vulnerabilities in the components that I have used in my application, I regularly check for updates and keep them up to date.

**A10:2017 - Insufficient Logging & Monitoring**

In professional environments, logging for traceability of events is important. The risk is that failed logins, high-value transactions or errors are insufficiently logged. An attacker spends a lot of time examining an application or system to find the vulnerabilities. If a system does not have sufficient logging and monitoring, the attacker can calmly search for errors and vulnerabilities. This increases the chance of successfully finding and exploiting a vulnerability. Ideally, the application has monitoring software that alerts the user to this threatening investigation. If not, at least one mechanism is needed to inform about the intrusion of an attacker.

Several steps could be undertaken to secure my application from insufficient logging & monitoring.

* All login, access control, and server-side input validation errors should be logged with sufficient user context to identify suspicious or malicious accounts. Logs should be retained for a period of time that allows delayed forensic analysis.
* Logs have to be created in a format that can be easily used by central log management tools.
* Establish or adopt an incident response and recovery plan.

# References

(2014). Retrieved from StackOverflow: https://stackoverflow.com/questions/24257336/owasp-top-ten-attacks-and-spring-security

*Broken Access Control*. (n.d.). Retrieved from HDIV: https://hdivsecurity.com/owasp-broken-access-control

*Components With Known Vulnerabilities*. (n.d.). Retrieved from HDIV Security: https://hdivsecurity.com/owasp-using-components-with-known-vulnerabilities

*Injection Theory*. (n.d.). Retrieved from OWASP Foundation: https://owasp.org/www-community/Injection\_Theory

*Insecure Deserialization*. (n.d.). Retrieved from crashtest-security: https://wiki.crashtest-security.com/insecure-deserialization

Morgenroth, S. (2020, 9 4). *Cross-site Scripting in React Web Applications*. Retrieved from netsparker: https://www.netsparker.com/blog/web-security/cross-site-scripting-react-web-applications/

*OWASP Top 10 2017*. (2017). Retrieved from OWASP Foundation: https://owasp.org/www-project-top-ten/2017/

*OWASP Top 10: Sensitive Data Exposure*. (2019, 11 1). Retrieved from DeepSource: https://deepsource.io/blog/owasp-top-ten-sensitive-data-exposure/

Rasokat, J. (2019, 2 14). *How to prevent Insufficient Logging and Monitoring*. Retrieved from medium: https://medium.com/@javan.rasokat/owasp-appsensor-logging-and-monitoring-2518712ee0fe

Shahnovsky, R. (2019, 7 3). *What is Security Misconfiguration and How to Avoid It* . Retrieved from Guardicore: https://www.guardicore.com/2019/03/understanding-and-avoiding-security-misconfiguration/

Spukas, L. (2020, 3 22). *Preventing XSS in React Applications* . Retrieved from dev: https://dev.to/spukas/preventing-xss-in-react-applications-5f5j

*What Is Broken Authentication?* (2020, 8 20). Retrieved from auth0: https://auth0.com/blog/what-is-broken-authentication/

*XXE Injection*. (n.d.). Retrieved from portswigger: https://portswigger.net/web-security/xxe