BSEP Projekat

Kontolna tačka 2

## Prepared statements

The database parses, executes, translates and then stores the statement without executing it.

Only once the application provides values for the statement, the values are bound to the statement and the result set retrieved.

Since query and untrusted inputs are provided at different times and in different contexts, the two don’t get ‘mixed’ and the untrusted input is unable to change the meaning of the original query.

<https://medium.com/bugbountywriteup/security-preventing-sql-injection-sqli-ade81c5fd092>

If you need to provide a sorting column in your web application you should implement a whitelist to validate the value of the order by, it should always be limited to something like 'firstname' or 'lastname' or 1,2 etc. Remember it is a combination of both, you should use both **always use** a **prepared statement** when dealing with SQL but **also use input validation** on parts of the query which are not seen as a dynamic query parameter when using a prepared statement.

<https://blog.jdriven.com/2017/10/sql-injection-prepared-statement-not-enough/> - order by, group by

## Output encoding

You might be thinking what if I do not generate HTML directly like shown in the example above and use some UI framework like JSP or Angular JS?

In that case, you might already be safe as many frameworks have built-in encoding while rendering user input. But you need to check your application either manually or by using some security testing tools like [OWASP ZAP](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project) or something similar.

(<https://www.linkedin.com/pulse/xss-prevention-using-output-encoding-neeraj-malhotra>)

In case of Angular JS, the ng-bind directive which is commonly used as {{ expression }} automatically HTML encodes the content.

**Angular Security**

### Angular’s cross-site scripting security model

To systematically block XSS bugs, Angular treats all values as untrusted by default. When a value is inserted into the DOM from a template, via property, attribute, style, class binding, or interpolation, Angular sanitizes and escapes untrusted values.

Sanitization is the inspection of an untrusted value, turning it into a value that's safe to insert into the DOM. In many cases, sanitization doesn't change a value at all. Sanitization depends on context: a value that's harmless in CSS is potentially dangerous in a URL. Angular sanitizes untrusted values for HTML, styles, and URLs.

Never generate template source code by concatenating user input and templates.

Don't generate Angular templates on the server side using a templating language; doing this carries a high risk of introducing template-injection vulnerabilities.

## HTTP-level vulnerabilities

Angular has built-in support to help prevent two common HTTP vulnerabilities, cross-site request forgery **(CSRF or XSRF)** and cross-site script inclusion (XSSI). Both of these must be mitigated primarily on the server side, but Angular provides helpers to make integration on the client side easier.

<https://angular.io/guide/security>

<https://ripcordsystems.com/2019/03/09/angular-how-to-prevent-xss-attacks-code-examples/>

<https://www.dotnettricks.com/learn/angular/tips-to-secure-your-angular-applications>

## Check your dependencies with SNYK

There’s a good chance you don’t know how many direct dependencies your application uses. It’s extremely likely you don’t know how many transitive dependencies your application uses. This is often true, despite dependencies making up the majority of your overall application. Attackers target open source dependencies more and more, as their reuse provides many victims for a malicious hacker. It’s important to ensure there are no known vulnerabilities in the entire dependency tree of your application.

[Snyk](http://snyk.io/) tests your application build artifacts, flagging those dependencies that have known vulnerabilities. It provides you with a list of vulnerabilities that exist in the packages you’re using in your application as a dashboard.

<https://developer.okta.com/blog/2018/07/30/10-ways-to-secure-spring-boot>

## Passwords

**BCrypt** has been out there since 1999 and does a better job at being GPU/ASIC resistant than PBKDF2 but I wouldn’t recommend it for new systems

<https://medium.com/analytics-vidhya/password-hashing-pbkdf2-scrypt-bcrypt-and-argon2-e25aaf41598e>

For managing passwords in general, we recommend using either SCrypt or Argon2.

**SCrypt** is a better choice today: better design than BCrypt (especially in regards to memory hardness) and has been in the field for 10 years.  
On the other hand, it has been used for many cryptocurrencies and we have a few hardware (both FPGA and ASIC) implementation of it.  
Even though they’re specifically for mining they can be repurposed for cracking.

**Conclusion**

In 2019 I’d recommend **not to use** PBKDF2 or BCrypt in the future and highly recommend Argon2 (preferrably Argon2id) for newer systems.

Scrypt can be a second choice on systems where Argon2 is not available, but keep in mind that it has the same issues with respect to side-channel leakage.

In short: use Argon2id if you can, use Argon2d in almost every other case, consider Argon2i if you really do need memory side-channel attack resistance. (<https://security.stackexchange.com/questions/193351/in-2018-what-is-the-recommended-hash-to-store-passwords-bcrypt-scrypt-argon2> )

* Prefer Argon2 over scrypt
* Prefer scrypt over bcrypt

<https://zaiste.net/security/vs/bcrypt-scrypt-argon2/>

## HTTPS

[*https://medium.com/@niral22/2-way-ssl-with-spring-boot-microservices-2c97c974e83*](https://medium.com/@niral22/2-way-ssl-with-spring-boot-microservices-2c97c974e83) *- microservices and https*

[*https://groups.google.com/forum/#!topic/dropwizard-user/1I\_litGtsCo*](https://groups.google.com/forum/#!topic/dropwizard-user/1I_litGtsCo) *- building truststore with keytool*

[*https://stackoverflow.com/questions/11617210/how-to-properly-import-a-selfsigned-certificate-into-java-keystore-that-is-avail*](https://stackoverflow.com/questions/11617210/how-to-properly-import-a-selfsigned-certificate-into-java-keystore-that-is-avail) *- how to import certificate to JVM*