Security Optimization of JSON Web Token (JWT) Authentication in Laravel Framework Against Cross Site Scripting (XSS) Threats

Iqbal Atma Muliawan*, Sri Supatmi

Master of Information Systems, Universitas Komputer Indonesia, Bandung, Indonesia

*Corresponding E-mail: iqbal.75124005@mahasiswa.unikom.ac.id

Abstract. This research aims to optimize the security of the Laravel JWT authentication library created by tymondesign. The security improvement focuses on complying with RFC 6749 by using distinct tokens for access and refresh, and on mitigating Cross-Site Scripting (XSS) attacks by preventing stolen access tokens from being reused in an attacker's browser. The approach to improve security is by separating the access token and refresh token in accordance with OAuth 2.0 as defined by RFC 6749, and by performing token binding on the access token through an access token verifier stored in an HttpOnly cookie. This token is used to verify that the access token being sent is valid from the same browser that issued it. The access token verifier stored in the HttpOnly cookie cannot be stolen via XSS attacks because HttpOnly cookies cannot be read by JavaScript. In this way, the access token remains secure, and even if stolen, it cannot be used because the attacker does not possess the access token verifier. The research was conducted using the Waterfall methodology with a proof-of-concept implementation and comparative testing against the original tymondesign/jwt-auth library. The results show that the proposed approach successfully mitigates the reuse of stolen access tokens in an attacker's browser and prevents stolen access tokens from being used multiple times to refresh tokens, because the new approach uses separate access and refresh tokens stored in HttpOnly cookies. This improvement contributes to more secure JWT authentication practices in RESTful APIs and provides insight for enhancing web security against XSS attacks.

Keywords: JWT, Authentication, XSS, Token Binding, Refresh Token

1. Introduction

The authentication process is the process used to verify a person's identity within a system based on the credentials they possess. This process is important because it serves as the main gateway to prevent unauthorized user from impersonating other users in unauthorized way, which could result in material or non-material losses. One of the system architectures that also requires authentication is the RESTful API. RESTful API is a standard for building web services that allows two or more systems to communicate through the HTTP protocol. RESTful API utilizes a token that is included with each request, as every request is stateless by design (Huang et al., 2015). JSON Web Token (JWT) can be used for RESTful APIs and stored in the client's browser, such as in *localstorage*, *session storage*, or cookies. However, storing an access token in local storage, session storage, or a cookie (without *HTTP-only* configuration) is vulnerable to Cross-Site-Scripting (XSS) attacks.

This research aims to optimize the security of the Laravel JWT authentication framework library created by tymondesign. The *tymondesign/jwt-auth* library generates only a single type of token, which is an access token. The access token is utilized to access protected endpoints, such as */api/users/profile* (to retrieve personal user information). Additionally, the same token is used to request a new token, effectively serving both as an access token and a refresh token. This approach does not comply with oauth2, which is defined in RFC 6749, that uses a refresh token to regenerate a new token. Using the same token to regenerate a new token introduces a new vulnerability. If attackers can manage to steal the token, they can repeatedly refresh it, thereby extending their unauthorized access without the user's knowledge. Therefore, this research proposes an improvement by adopting the OAuth 2.0 approach, which utilizes two types of tokens: access token and refresh token(Flanagan, 2024)(*The OAuth 2.0 Authorization Framework*, 2012). Since access tokens are short-lived and periodically rotated through a refresh token, a stolen access token can only be misused for a limited time. With a separate access token and refresh token, users can rotate the token without requiring user interaction (re-authentication, which involves sending the user's credentials again). Refresh token store in a cookie with an *httpOnly* configuration, which makes this token secure from XSS attacks because attackers cannot access the cookie through JavaScript (Kwon et al., 2020). Nevertheless, this approach is not entirely foolproof, as there remains a potential vulnerability whereby an attacker who successfully obtains the access token could still perform harmful actions within seconds, such as initiating unauthorized balance transfers before the token expires. At this point, the second enchantment proposed in this study is introduced. Instead of relying on a single token, the system generates three distinct tokens: an access token, a refresh token stored with *httpOnly* attributes, and an access token verifier, also stored as *httpOnly*. The access token verifier makes sure that this token is used by same browser that issued access token. This can mitigate risk when access token stolen by attacker via XSS attack

2. Literature Review

2.1 Authentication

Authentication is the process of verifying a user's identity before granting access to system resources. There are three main approaches to authentication: something the user knows (e.g., username, PIN, or password), something the user has (e.g., ATM card, NFC device, or authenticator app), and something the user is (biometrics such as fingerprint, face recognition, retina scan, or voice recognition) (Bucko et al., 2023)(MacIej et al., 2019). Authentication is crucial in preventing unauthorized individuals from impersonating legitimate users and performing malicious actions (Kimani et al., 2023).

2.2 JWT (JSON Web Token)

JWT (JSON Web Token) is a standard defined in RFC 7519 (Jones, 2015) that provides a URL-safe format for transmitting claims between two parties using encoded JSON. A JWT consists of three components: the header, the payload, and the signature(Alkhulaifi & M. El-Alfy, 2020). The header contains the algorithm used, the payload contains the data being transferred, and the signature is used to verify the token's validity, ensuring the server issued it and has not been modified (Adam et al., 2020). JWT is not an encryption mechanism; therefore, anyone with access to the token can view its content. As a result, it is not recommended to store sensitive information in the payload to prevent data leakage.

2.3 RFC 6749

RFC 6749 is a standard defined for the authorization framework. This model can be used for client-server-based authentication to protect access to protected resources. The standard utilizes two types of tokens: access tokens and refresh tokens(*The OAuth 2.0 Authorization Framework*, 2012). Access tokens have a short lifetime, while refresh tokens have a longer lifetime. Access tokens are used to access protected resources, whereas refresh tokens are used to regenerate new access tokens. This mechanism can be effectively utilized for token rotation.

2.4 XSS (Cross-Site Scripting)

XSS (Cross-Site Scripting) is an attack technique that involves injecting JavaScript code to be executed in the victim's browser. This script can be used to steal all data stored on the client-side, such as localstorage and cookies (unless configured with the httpOnly flag)(Gupta & Gupta, 2017). Tokens accessed by the script can then be sent to the attacker's server and later used to impersonate the victim on the web system(Johari & Sharma, 2012). XSS cannot access cookies that are configured with the httpOnly flag, as JavaScript is unable to read them. However, a drawback of using httpOnly cookies for accessing protected resources is that they are vulnerable to CSRF (Cross-Site Request Forgery) attacks, as the token is automatically sent with every request to the server (Rankothge & Randeniya, 2020).

3. Method

3.1 Research Design

This research is a development-oriented study that focuses on enhancing the security of an existing library. The type of research conducted falls under software engineering research, utilizing a research and development (R&D) approach (Gustiani & Sriwjaya, 2019). This approach was chosen because the study aims to design, develop, and test a software-based solution, rather than relying on user opinions or feedback. The emphasis is placed on the technical aspects of system improvement through a structured development approach.

The software development methodology applied in this research is the Waterfall model, consisting of sequential stages: system requirement analysis, design system, implementation, testing, release. This method was chosen over iterative approach such as Agile or Spiral because of system requirement already defined. Since the primary objective was to propose and evaluate a specific improvement (enhanced XSS protection and refresh token mechanism), the development does not require user feedback or dynamic requirement changes. The Waterfall model provides a sequential development flow that aligns with the nature of this study, where each stage is completed before proceeding to the next (Heriyanti & Ishak, 2020). This study focuses on proof of concept, not user-centered development and specification to proof the concept already defined, that to protect token from XSS attack so when hacker stole access token, they cannot use that token to impersonate as victim in a system.

The testing phase conducted in this research focused on the technical validation. User feedback was not involved in the testing process, as it was not necessary to answer the research hypothesis. The central hypothesis of this research is that XSS attacks on RESTful API-based systems can be prevented by introducing an access token verifier securely stored in a cookie with the *httpOnly* configuration. The following stages of the Waterfall model are presented in this study, as shown in Figure 1.

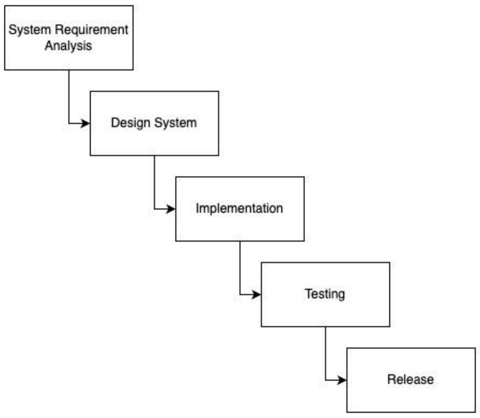

Figure 1. Waterfall Method

Here is detail of 5 waterfall stages model that used in this study:

1. System Requirement Analysis: System requirement is a process of gathering information about library requirements. These requirements outline the expected functionality of the library. They serve as a reference for system design and implementation, as they define the system's boundaries.
2. System Design: This stage focuses on technical planning to determine what actions can be taken to meet the library's expectations. The planning includes the basic jwt-auth and two enhancement steps.
3. Implementation: Implementation is the process of writing the improvement program for the tymondesign/jwt-auth library. The developed program is based on that library and is rewritten to produce a new version as a separate library.
4. Testing : There will be two testing phases. The first test aims to verify the token rotation process using a refresh token without requiring the user to re-enter their credentials. The second test simulates the theft of an access token through an XSS attack. The attacker will then use the stolen token in their browser. The expected output of this stage is that the stolen token cannot be used.
5. Release: At the release stage, the library will be made publicly available through GitHub.

3.1 Technical Approach

This research employs two technical approaches. The first approach is to separate the access token and refresh token. The second approach is to implement token binding through an access token verifier using the *httpOnly* configuration. Architecture design of technical approach that proposed in this study shown in Figure 2 :

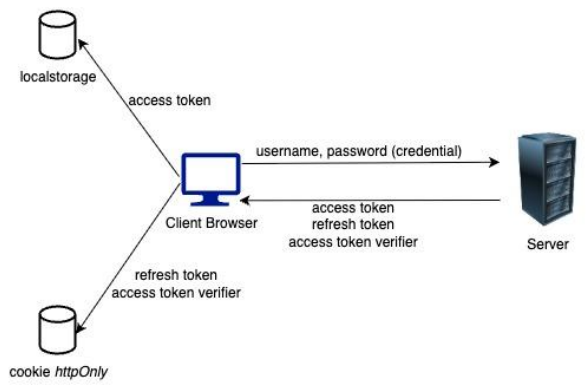

Figure 2. Architecture Design

1. Separate the access token and refresh token

This approach creates new type of token called refresh token. Refresh token is used to generate new pair of access token and refresh token and invalidated old token. Access token only used to access protected resources and cannot be used to generate new tokens. Access token stored in localstorage and refresh token stored in cookie with httpOnly configuration. Access token will have a shorter Time to Live (TTL), around 10-15 minutes while refresh token has longer TTL, between 3-7 days. This setup designed to reduce the attacker opportunity if access token is stolen and make sure that the access token becomes invalid after few minutes. By separating access token and refresh token, we can perform token rotation without user interaction. For example when access token expires, but user still interacting with system, the refresh token can be used to generate new pair of access token and refresh token before system consider it logout. The refresh token flow is illustrated in Figure 3:

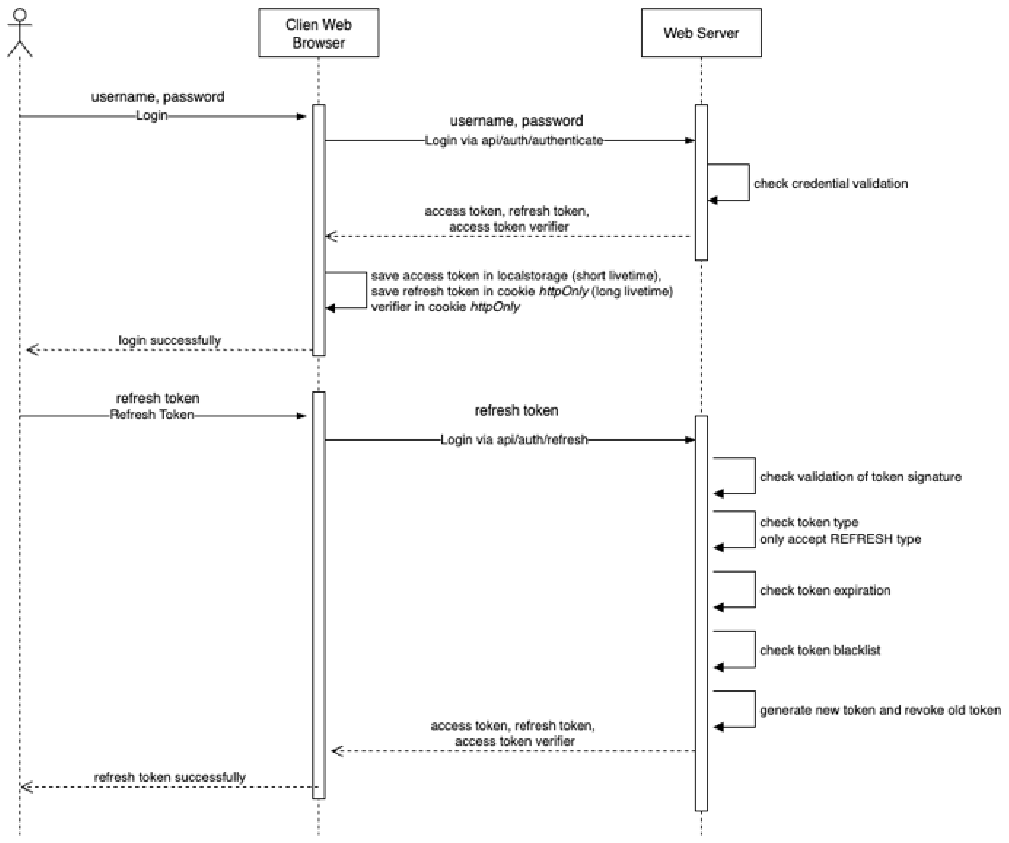
Clien Web Browser
Web Server
username, password
Login
username, password
Login via api/auth/authenticate
check credential validation
access token, refresh token, access token verifier
save access token in localstorage (short livetime), save refresh token in cookie httpOnly (long livetime) verifier in cookie httpOnly
login successfully
refresh token
Refresh Token
refresh token
Login via api/auth/refresh
check validation of token signature
check token type only accept REFRESH type
check token expiration
check token blacklist
generate new token and revoke old token
access token, refresh token, access token verifier
refresh token successfully

Figure 3. Approach 1 (separation of access and refresh token)

2. Access token verifier

This approach create new kind of token that called access token verifier that stored in httpOnly cookie. When user authentication successfully, system will generate access token and access token verifier that use random string. That random string then hashed and set as JWT claim in access token with key atv (access token verifier). The plain text of access token verifier stored in cookie httpOnly. Access token verifier flow is illustrated in Figure 4 :

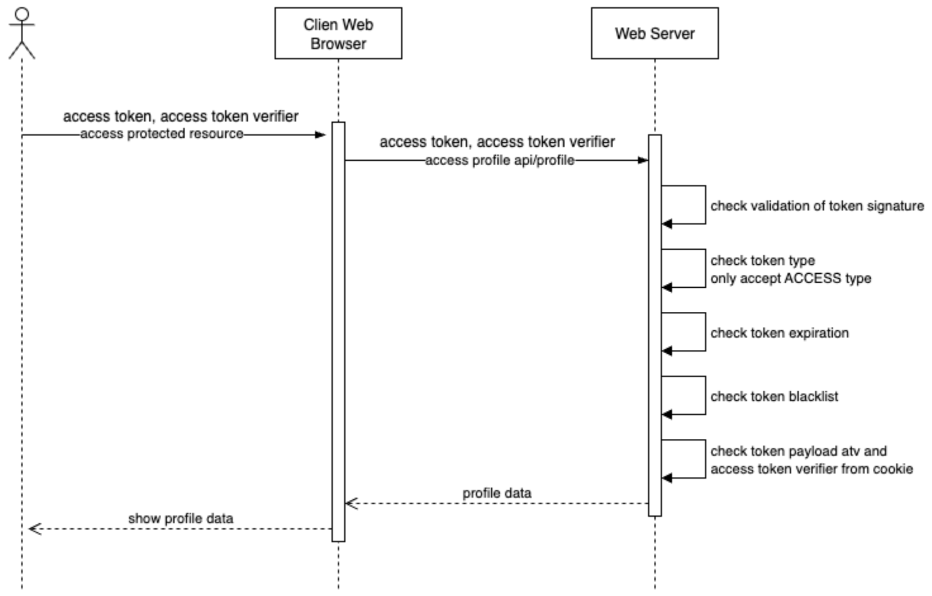
Clien Web Browser
Web Server
access token, access token verifier
access protected resource
access token, access token verifier
access profile api/profile
check validation of token signature
check token type
only accept ACCESS type
check token expiration
check token blacklist
check token payload atv and
access token verifier from cookie
profile data
show profile data

Figure 4. Approach 2 (access token verifier)

The following are the steps in the token binding authentication check process:

1. When attempting to access a protected resource, the client sends both the access token and the access token verifier to the server.
2. The server checks the access token verifier and compares it with the hashed atv value stored in the payload of the access token.
3. If the access token verifier is not sent, or if the hashed value in the payload does not match the value stored in the *httpOnly* cookie, the system considers the token to have been compromised. As a result, the access token will be deactivated or blacklisted.

4. Results and Discussion

The *tymondesign/jwt-auth* library has two main weaknesses. First, when the access token expires, the user cannot perform a token refresh and must request a new token by resubmitting their credentials. This limitation exists because the refresh process uses the same access token that is also used to access protected resources. The second weakness is related to security. If an XSS attack occurs and the token is successfully stolen, the attacker can use the stolen token to impersonate the victim. The weaknesses found in the *tymondesign/jwt-auth* library were addressed and developed into a new library. The improved library can be found at https://github.com/iqbalatma/laravel-jwt-authentication under the name *iqbalatma/laravel-jwt-authentication*. This library is open source, and its source code is publicly available. It can also be installed via Composer, as it has been published on Packagist.

With the new approach, two key improvements were made based on the *tymondesign/jwt-auth* library. The first improvement is the implementation of a token rotation mechanism. This token rotation mechanism allows users to continue accessing the system after the access token expires by using a refresh token, without needing to resend their credentials. The second improvement is the addition of an access token verifier, which is stored in an *httpOnly* cookie and checked against the atv key in the access token payload, where the value is stored in hashed form. With this approach, even if an attacker successfully steals the access token and obtains the hashed atv value from the payload, they cannot determine the original verifier value. They thus cannot forge the access token verifier stored in the cookie. This method significantly enhances security and minimizes the impact of XSS attacks on the victim. Tables 1 and 2 present the comparison of testing results before and after the improvements were implemented:

Table 1 Scenario test comparation

Test Scenario	**Expected Result**	**Actual Result on iqbalatma/laravel-jwt-authentication**	**Status**	**Actual Result on tymondesign/jwt-auth**	**Status**
Access token expiration	User can get new token without re-login	Can get new access token using refresh token rotation	Pass	Requires re-login	Fail
Refresh token usage	Refresh token can generate new token pairs (access and refresh)	Supported (separate access and refresh token)	Pass	Not supported	Fail
Token rotation	Old access token invalidated after refresh	Old access token invalidated	Pass	Old access token invalidated	Pass
Logout	Access token invalidated	Access token invalidated	Pass	Access token invalidated	Pass
Token Device Isolation	Access token can be used only by device that invoke token	Access token only can be used by device that invoke token	Pass	Access token can be used anywhere	Fail
Token Storage Location	Support store token for refresh in httpOnly cookie to prevent XSS	Supported store in httpOnly	Pass	Not supported	Fail
Impersonate user using stolen access token via XSS	Stolen token cannot be used by hacker	Stolen access token cannot be used to impersonate victim in hacker device	Pass	Stolen access token can be used to impersonate victim in hacker device	Fail
Modified jwt claim	Modified jwt claim should be rejected	Reject modified jwt claim due to invalid token signature	Pass	Reject modified jwt claim due to invalid token signature	Pass
Reject expired token	Expired token should be rejected	Expired token rejected	Pass	Expired token rejected	Pass

The comparison results between the new approach implemented in the ***iqbalatma/laravel-jwt-authentication*** library and the ***tymondesign/jwt-auth*** library can be seen in Table 3:

Table 2 Comparison results between 2 libraries

Aspect	tymondesign/jwt-auth	iqbalatma/laravel-jwt-auth
Token rotation	Uses same token for both access and refresh purposes.	Separate access and refresh token for access and refresh purposes
Token binding support	Does not support token binding	Support token binding on access token
Token lifetime	Only use single access token with single lifetime	Can use short lifetime for access and longer lifetime for refresh token
Token storage location	Access token can store in localstorage or cookie without httpOnly attribute	Access token can store in localstorage or cookie without httpOnly attribute and refresh token can store in cookie with httpOnly attribute.
XSS Resistant	Stolen access token can be reused to impersonate victim	Stolen access token cannot be reused to impersonate victim because there is access token verifier for token binding
Implementation complexity	Does not require auto refresh token mechanism since using single access token	Need to implement auto refresh token without user interaction when access token expired
Security overhead	Middleware process only checks for token validity, expires time, and blacklist (lower CPU usage but vulnerable to XSS)	Middleware process checks for token validity, expires time, blacklist, avt existence, avt and hashing validity (increase CPU but resistant to XSS)
Trade-off	Faster performance with weaker security	Stronger security with slight performance cost

5. Conclusion

This study demonstrates that the two improvement approaches—namely, the separation of access and refresh tokens, and the implementation of access token binding using an access token verifier—have effectively increased the security of the library and reduced the impact of XSS attacks. Even if an attacker manages to steal an access token through an XSS attack, the token cannot be used to access protected resources because the system requires verification through an access token verifier stored in a cookie with *httpOnly* configuration, which is not accessible via JavaScript. Additionally, the separation of tokens enables secure and continuous token rotation without compromising the user experience. These improvements provide a practical and safer approach to handling authentication in web applications.

References

Adam, S. I., Moedjahedy, J. H., & Maramis, J. (2020, October 27). RESTful Web Service Implementation on Unklab Information System Using JSON Web Token (JWT). *2020 2nd International Conference on Cybernetics and Intelligent System, ICORIS 2020*. https://doi.org/10.1109/ICORIS50180.2020.9320801

Alkhulaifi, A., & M. El-Alfy, E.-S. (2020). *Exploring Lattice-based Post-Quantum Signature for JWT Authentication: Review and Case Study*. IEEE. https://doi.org/10.1109/VTC2020-Spring48590.2020.9129505

Bucko, A., Vishi, K., Krasniqi, B., & Rexha, B. (2023). Enhancing JWT Authentication and Authorization in Web Applications Based on User Behavior History. *Computers*, *12*(4). https://doi.org/10.3390/computers12040078

Flanagan, H. (2024). Token Lifetimes and Security in OAuth 2.0: Best Practices and Emerging Trends. *IDPro Body of Knowledge*. https://doi.org/doi.org/10.55621/idpro.108

Gupta, S., & Gupta, B. B. (2017). Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. In *International Journal of System Assurance Engineering and Management* (Vol. 8, pp. 512–530). Springer. https://doi.org/10.1007/s13198-015-0376-0

Gustiani, S., & Sriwjaya, P. N. (2019). RESEARCH AND DEVELOPMENT (R&D) METHOD AS A MODEL DESIGN IN EDUCATIONAL RESEARCH AND ITS ALTERNATIVES. *HOLISTICS JOURNAL*, *11*(2). ISSN: 2657-1897.

Heriyanti, F., & Ishak, A. (2020). Design of logistics information system in the finished product warehouse with the waterfall method: Review literature. *IOP Conference Series: Materials Science and Engineering*, *801*(1). https://doi.org/10.1088/1757-899X/801/1/012100

Huang, X. W., Hsieh, C. Y., Wu, C. H., & Cheng, Y. C. (2015). A token-based user authentication mechanism for data exchange in RESTful API. *Proceedings - 2015 18th International Conference on Network-Based Information Systems, NBiS 2015*, 601–606. https://doi.org/10.1109/NBiS.2015.89

Johari, R., & Sharma, P. (2012). A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection. *Proceedings - International Conference on Communication Systems and Network Technologies, CSNT 2012*, 453–458. https://doi.org/10.1109/CSNT.2012.104

Jones, M. (2015). *Internet Engineering Task Force (IETF)*. http://www.rfc-editor.org/info/rfc7519.

Kimani, C., Obuhuma, J. I., & Roche, E. (2023). Multi-Factor Authentication for Improved Enterprise Resource Planning Systems Security. *International Journal of Information Technology and Computer Science*, *15*(3), 42–54. https://doi.org/10.5815/ijitcs.2023.03.04

Kwon, H., Nam, H., Lee, S., Hahn, C., & Hur, J. (2020). (In-)Security of Cookies in HTTPS: Cookie Theft by Removing Cookie Flags. *IEEE Transactions on Information Forensics and Security*, *15*, 1204–1215. https://doi.org/10.1109/TIFS.2019.2938416

MacIej, B., Imed, E. F., & Kurkowski, M. (2019). Multifactor Authentication Protocol in a Mobile Environment. *IEEE Access*, *7*, 157185–157199. https://doi.org/10.1109/ACCESS.2019.2948922

Rankothge, W. H., & Randeniya, S. M. N. (2020). Identification and Mitigation Tool for Cross-Site Request Forgery (CSRF). *IEEE Region 10 Humanitarian Technology Conference, R10-HTC*, *2020-December*. https://doi.org/10.1109/R10-HTC49770.2020.9357029

The OAuth 2.0 Authorization Framework. (2012). http://www.rfc-editor.org/info/rfc6749.