File Upload Security: Essential Practices for Programmers

Anrie Suryaningrat[*1], Desi Ramayanti[2], Glen Maxi Taberima[3], Panji Pratama Kurniawan[4]
[1,2,3] Department Informatics Engineering, Faculty Engineering and Informatics, Dian Nusantara University, Indonesia
[4]Information Systems Faculty of Science and Technology, University of Raharja, Indonesia
E-mail: [*1]**anrie.suryaningrat@dosen.undira.ac.id**, [2]desi.ramayanti@undira.ac.id, [3]411192017@mahasiswa.undira.ac.id, [4]panji.pratama@raharja.info

Abstract

File upload processes, while convenient, introduce data security risks that must be addressed. Potential malware infections, sensitive information leaks, and data corruption necessitate the implementation of effective strategies and solutions. Data integrity refers to the accuracy, consistency, and reliability of stored and processed data. Human error, system failures, and cyberattacks can compromise data integrity, necessitating effective preventive and mitigation measures. Data security and file upload integrity are critical aspects of modern web applications. Given the escalating cyber threat landscape, implementing best practices to safeguard user data and ensure the authenticity of uploaded files is paramount. This paper delves into the implementation of best practices for protecting user data and ensuring file upload authenticity. The study proposes the implementation of best practices for secure file upload mechanisms in web applications, including the innovative application of hash-salting techniques. By adhering to the guidelines presented, developers can bolster the defenses of their web applications against a wide range of cyber threats, enhance user trust, and protect sensitive data. This approach also serves as an effective solution for improving efficiency, security, and compliance in digital records management.

Keywords *— Data Security, Data Integrity, File Upload, Web Application*

1. INTRODUCTION

The proliferation of web applications has witnessed remarkable growth over the past decade. Statista data reveals that the global market share of web applications has soared from 40% in 2013 to 65% in 2023. This surge is attributed to several factors, including advancements in web technologies such as HTML5, CSS3, and JavaScript, enabling developers to create more feature-rich and responsive web applications; the increasing prevalence of mobile devices; and the widespread availability and accessibility of the internet at affordable rates across the globe, empowering more individuals to utilize web applications.

While web applications offer a plethora of benefits, they also introduce a range of security challenges and vulnerabilities, particularly concerning file upload functionalities. File upload features are integral components of many web applications, allowing users to upload various file types, including images, documents, and videos. File upload vulnerabilities can be defined as security loopholes that enable attackers to upload malicious files to a web application's server (Al-Khannak & Nehal, 2023). These malicious files can be executed to compromise servers, steal data, or even disrupt web applications (Chen et al., 2024). A prime

example is the Magecart malware, which empowers hackers to pilfer customer payment information, including credit card numbers and personal details (Mathew, 2021)

Several factors can potentially lead to file upload vulnerabilities:

1. Unvalidated File Types: Web applications that fail to validate the type of uploaded files allow attackers to upload malicious files with modified extensions (Hilmi & Yunan, 2022; Sharif, 2022). File integrity verification is crucial for ensuring data authenticity and accuracy, especially for sensitive data.
2. Excessive File Size Limits: Overly generous file size limitations can enable attackers to upload large files containing malware, backdoors, or other malicious code that can overwhelm systems (Hilmi & Yunan, 2022; Upadhyay & Ware, 2023).
3. Absence of Virus and Malware Scanning: The lack of virus and malware scanning mechanisms for uploaded files can allow attackers to infect servers and user devices (Lala et al., 2021).
4. Insecure File Storage: Storing uploaded files in unsecured and unencrypted locations can facilitate unauthorized access or grant "execute" permissions to upload directories, potentially leading to path-traversal attacks (Chawda et al., 2021; Lala et al., 2021; Sharif, 2022).
5. Outdated Web Application Software: Outdated web application software may contain known vulnerabilities that attackers can exploit (David Odera et al., 2023; Ibrahim et al., 2019; Kumar & Rani, 2022; R. Fujdiak et al, 2019).

Data integrity, a fundamental element of the Confidentiality, Integrity, Availability (CIA) Triad, plays a critical role in ensuring data accuracy and reliability (Chai & Zolkipli, 2021). Data with preserved integrity must remain free from manipulation, modification, or unauthorized access, safeguarding the trustworthiness and reliability of information. Notably, database attacks like SQL injection can occur not only due to confidentiality issues but also integrity and availability concerns (Alghawazi et al., 2022).

Without data integrity, cybersecurity risks escalate dramatically. Cybercriminals can alter data for personal gain, tarnish reputations, or disrupt operations. This can lead to severe consequences for organizations, including financial losses, operational disruptions, and reputational damage. File integrity verification is paramount for ensuring data authenticity and accuracy, particularly for sensitive data (David Odera et al., 2023). Therefore, implementing appropriate and sustainable strategies can help organizations minimize risks and enhance resilience against diverse cyber threats.

2. RESEARCH METHOD

The research flow is carried out as shown in Figure 1 below:

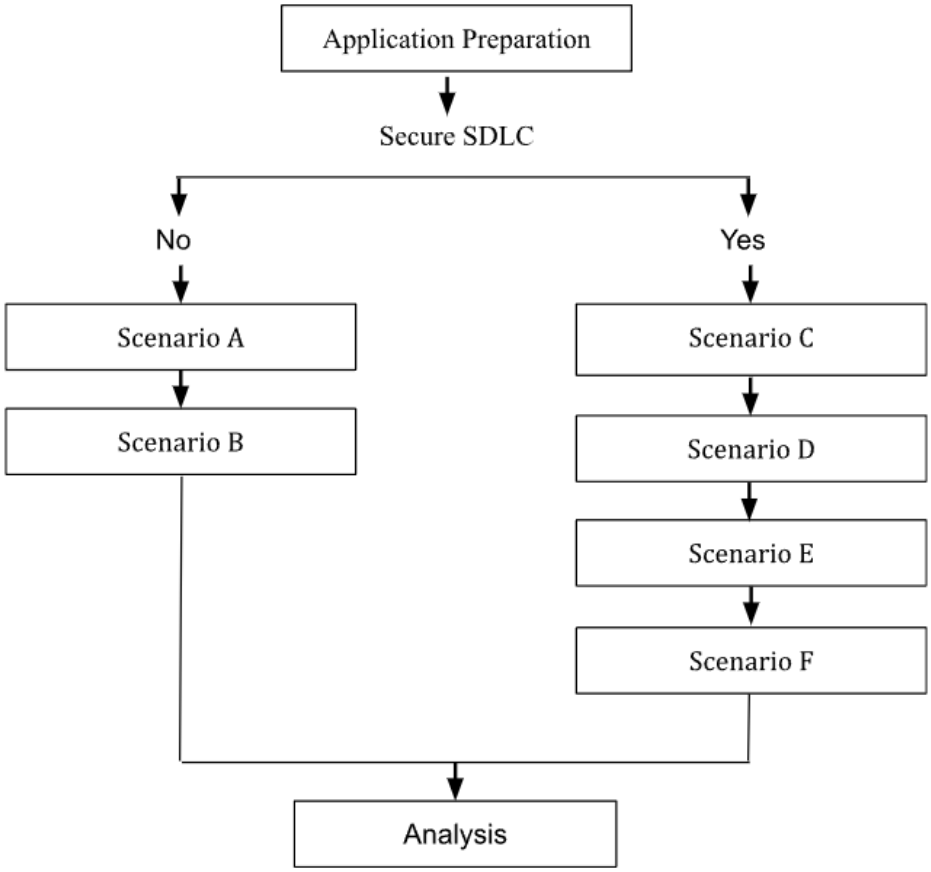

Figure 1. Research Flowchart

2.1. Test Application Preparation

This research employs two (2) Virtual Machines (VMs), each for the application server and database server, running on Ubuntu Server 20.04 LTS. The host system operates on Ubuntu Linux 20.04 LTS. The application server utilizes NginX as the web server and a test application developed using the CodeIgniter 3.1.x framework and PHP version 7.4.x. MariaDB serves as the database management system.

2.2. Testing

As illustrated in Figure 1, testing is conducted by dividing scenarios into two groups based on the implementation of Secure SDLC in the test application. The test group without Secure SDLC implementation begins with Scenario A and proceeds to Scenario B. In contrast, the test group with Secure SDLC implementation starts with Scenario C and continues to Scenario F. Testing focuses on image files with .jpg and .png extensions and predetermined sizes. Web shell, pdf, and text files are used as comparators.

Scenario testing involves parameter validation, verification, and custom salt-hashing, along with observing how data is stored in the database and files are stored on the storage medium. Scenario testing is performed with the following variations:

1. Scenario A: Files are uploaded without enabling any validation on either the presentation layer (file upload feature) or application layer. Files are directly stored in the storage directory without any permission restrictions. The storage of uploaded file data in the database consists of the file name and upload time.
2. Scenario B: Similar to Scenario A, but with the addition of enabling extension type validation on the presentation layer.
3. Scenario C: Similar to Scenario B, but with limiting and enabling validation of allowed extension types, as well as validation of content-type headers and file size on the application layer. Permission restrictions are applied to the storage directory.
4. Scenario D: All features of Scenario C with the addition of activating filtering mode, which involves cleaning/sanitizing file names from all control and unicode characters, validating the temporary file upload location and prohibiting execution access to the file upload location and limiting file names to no more than 255 characters. A unique code (featfile) is generated in the form of a custom hash-salt.
5. Scenario E: Implementation of Scenario D with the addition of custom hash-salt generation, file extension separation, and storage of file extensions as a separate attribute in the database.
6. Scenario F: Similar to Scenario E, this scenario involves resizing (for image files) according to the target media (responsive design), separating the unique uploader code and upload time, generating a new file name (Sharif, 2022),and setting the file name to all lowercase letters. In this scenario, once the file is validated, the original file is deleted from the server to avoid increasing the storage media load. The modeling for Scenario F is depicted in Figure 2.

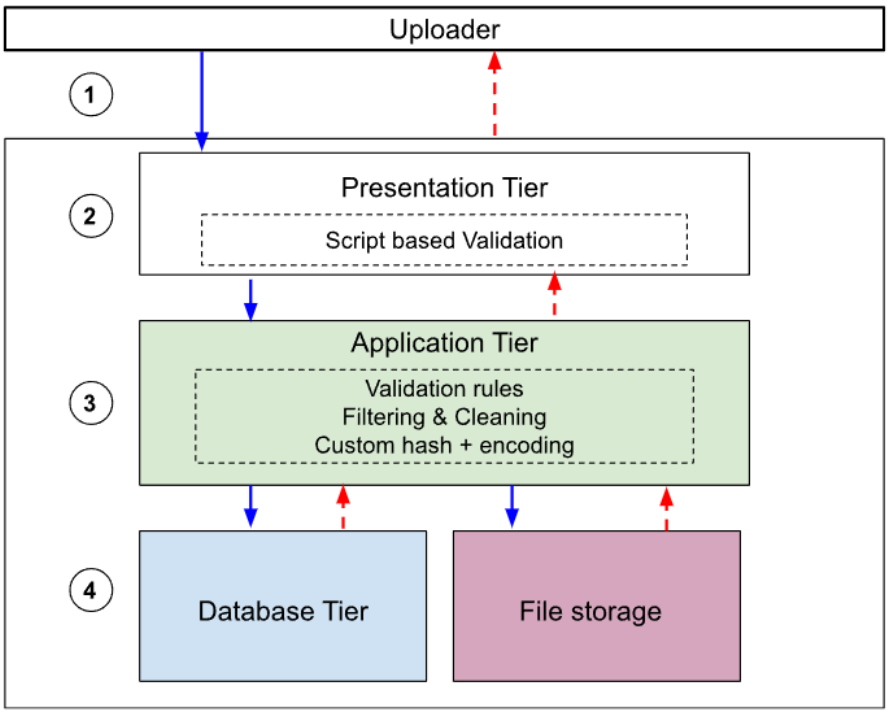

Figure 2. Scenario F Test

Detailed explanation of the stages in Figure 2:

Stage 1 and 2: File Upload Validation

These initial stages involve client-side validation using JavaScript to check the uploaded file's size and extension directly within the application's file upload feature. This validation is crucial in preventing the upload of malicious files disguised as image files. During this stage, not only image files but also web shell files with extensions like .sh.png or .py.jpg are uploaded to test the effectiveness of the file upload feature's validation mechanisms.

Stage 3: Application Layer Filtering

Stage 3 takes place on the application layer, where a series of filtering steps are implemented to further enhance security:

3-1. Upload Location/Path Validation:

The uploaded file's path is validated to ensure it adheres to the allowed upload locations and prevent unauthorized uploads.

3-2. Re-validation of File Extension and Content-Type Header:

The file's extension and content-type header are re-validated to ensure they match the expected image file format. This helps detect attempts to upload non-image files or files with modified extensions. If the extension type violates the allowed criteria, the upload process is immediately terminated.

3-3. File Size Validation:

The file size is checked against the defined maximum limit. If the file exceeds the permissible size, the upload process is halted at this sub-stage.

Sub stage code examples 3-1 through 3-3 as shown in Figure 3.

```
//moved into config
$config['upload_path']      = './assets/berita/' . $this->uri->segment(3) .'/';
$config['allowed_types']    = 'jpg|png';
$config['max_size']         = '5048';
$config['overwrite']        = TRUE;
$config['file_ext_tolower'] = TRUE;

$this->upload->initialize($config);
```

Figure 3. Validation of Extension, Upload Location, File Size, Renaming of Files to Lowercase

3-4. File Name Cleaning/Sanitization:

For files that pass the previous sub-stages, their names are sanitized to remove any invalid or inappropriate characters. This ensures adherence to naming conventions and prevents potential security vulnerabilities.

Sub stage code examples 3-4 as shown in Figure 4 below.

```
//featfile - tambahan untuk filter and cleaning
$search = array(" - "," ","'",".");
$replace = array('-','-','','');
```

Figure 4. Added Filter Mode and Filename Cleaning

3-5. Custom Hash-Salt Generation:

A unique hash-salt value is generated for the file using a custom hashing algorithm. This hash serves as a digital fingerprint for the file, aiding in file identification and integrity verification.

3-6. Extension Conversion and Separation:

The file extension is converted to lowercase and separated from the filename. This standardization ensures consistent handling of file extensions.

3-7. Image Resize (for Image Files):

For image files, the size is resized according to the target media requirements, typically using responsive design principles. This optimizes image display while reducing storage space.

3-8. Unique File Identifier Generation (featfile):

A unique file identifier (featfile) is created by combining the custom hash-salt value with 64-bit encoding. This unique identifier serves as a distinct signature for the file.

3-9. Uploader and Upload Time Separation:

The uploader's unique identifier and the upload timestamp are extracted from the data. This information is essential for tracking file uploads and maintaining audit trails.

3-10. New Filename Generation:

A new filename is generated based on specific criteria, such as the article title. The filename is formatted in lowercase letters for consistency.

Sub stage code examples 3-4 through 3-10 as shown in Figure 5.

```
$data_upload_files = $this->upload->data();

$foto_hash = random_string('alnum',11) . 'my' . random_string('nozero',4) . 'devio'.
     random_string('nozero',6) . 'rie'. random_string('numeric',20) . $this->input->
    post('created_by') . random_string('nozero',9) . strtotime($this->input->post('
    created_on')) . random_string('numeric',27);

$nmfile = $foto_hash . '-' . $this->input->post('slug', TRUE);

$foto_ext = substr( $data_upload_files['file_ext'], strrpos( $data_upload_files['
    file_ext'], '.' ) + 1 );

//do resizing
$image_sizes = array(
    'thumb'     => array(200, 100),
    'phablet'   => array(300, 200),
    'tablet'    => array(500, 400),
    'large'     => array(800, 600),
);

foreach ($image_sizes as $resize) :
    $config = array(
        'source_image' => $data_upload_files['full_path'],
        'new_image' => $this->upload->upload_path . $nmfile . '-'. $resize[0] . '-'
            . $resize[1] . '.' . $foto_ext,
        'maintain_ratio' => true,
        'width' => $resize[0],
        'height' => $resize[1]
    );

    $this->image_lib->initialize($config);
    $this->image_lib->resize();
    $this->image_lib->clear();

endforeach;
```

Figure 5. Formation of Unique File Identification Codes, Resizing Images, Separation of File Extensions

Stage 4 and 5: Database and Storage Operations

These final stages involve storing the processed file information and the file itself in the database and storage medium:

4-1. Database Storage:

The unique identifier, file extension, uploader's unique identifier, and upload timestamp are stored as separate attributes in the database. This structured data facilitates efficient retrieval and management of uploaded files.

4-2. File Storage:

Files that successfully pass all validation and filtering stages are stored in the designated storage medium, typically a file server. This ensures secure and accessible storage of the uploaded content.

Sub stage code examples 4-1 as shown in Figure 6.

```
$data_foto = array(
    'featimg'           => base64_encode($foto_hash),
    'featimg_ext'       => $foto_ext,
);

unlink($data_upload_files['full_path']);

$data = array_merge( $data_common, $data_foto );
```

Figure 6. Preparation of Storing Unique File Codes and Extensions as Fields for Each Attribute in the Database

The comprehensive validation and filtering processes implemented across these stages significantly enhance the security of file uploads in the web application. By rigorously checking file attributes, sanitizing file names, generating unique identifiers, and enforcing strict storage policies, the application effectively mitigates the risks associated with malicious file uploads.

2.3. Analysis

The test results will be analyzed by conducting comparisons between each scenario designed previously. This comparative analysis will focus on evaluating the effectiveness of the implemented security measures in preventing malicious file uploads and ensuring data integrity.

3. RESEARCH RESULTS AND DISCUSSION

3.1. Research Findings

This research delves into the effectiveness of employing layered validation and verification procedures for file upload functionalities in web applications. The study aims to demonstrate how these measures can enhance security and safeguard data integrity. The conducted testing serves as a testament to the implementation of Secure SDLC best practices and provides valuable insights for web application development.

Table 1. Comparison Between Test Scenarios

Scenario	Features	Secure SDLC
A	No	No
B	Validate Presentation layer (HTML DOM)	No
C	• Scenario B Feature plus • Validation at the Application Layer • Filtering File extension • Filtering Content Header-Type	Yes
D	• Scenario C features plus • Upload directory validation • Execution access restrictions on the upload directory • Sanitize the file names of all control characters and unicode	Yes
E	• Scenario D features plus • File extensions are stored as individual attributes in the database	Yes
F	• Scenario E features plus • Custom hash-salt • Max file size • Image recapture • File resizing • Fingerprint identification for each resizing • Only save resizing image(s) • Uploader ID record • Upload timestamp record • Unique ID (filefeat) • File re-name • File overwriting • Original source file never uploaded to storage	Yes

4. CONCLUSION

4.1. Comparison between scenarios

An analysis of the test results obtained from the pre-defined scenarios is presented as follows:

Scenarios A and B: Lack of Secure SDLC Implementation

Scenarios A and B represent the absence of proper Secure SDLC implementation during file upload functionality development. This results in significant security vulnerabilities:

1. Client-Side Validation Ineffectiveness: Relying solely on client-side JavaScript validation for file extensions on the presentation layer proves ineffective. Malicious actors can easily manipulate extensions to bypass validation and upload harmful files.
2. Path Traversal Attack Risk: The lack of proper access control permissions for the storage directory exposes the application to path traversal attacks. Attackers can exploit vulnerabilities to gain unauthorized access to arbitrary files on the server (Sharif, 2022).
3. Data Integrity Breaches: The absence of comprehensive validation and sanitization mechanisms leaves the application vulnerable to data integrity breaches. Malicious files can be uploaded, potentially compromising sensitive data or corrupting the application's functionality.

Scenario C: Enhanced Validation but Insufficient for Production

Scenario C introduces additional validation measures on the application layer, including extension type and content header-type checks. This improves security compared to Scenarios A and B:

1. Extension and Content-Type Validation: The application layer validation effectively detects and blocks attempts to upload malicious files disguised as image files by checking extensions and content headers. For instance, if a web shell file's extension is changed from .sh to .sh.jpg or a Python script's extension is changed from .py to .py.png, the upload process will be halted.
2. Unicode and Control Character Threats: However, this scenario remains vulnerable to attacks involving file names containing unicode or control characters. Attackers can exploit these characters to inject malicious code or bypass validation.

Scenario D: Comprehensive Security Mechanisms

Scenario D builds upon Scenario C by incorporating additional security mechanisms:

1. File Name Sanitization: File names are sanitized to remove control and unicode characters, preventing code injection attempts and ensuring consistent naming conventions.
2. Upload Location Validation: Upload location validation ensures that files are only processed when uploaded through authorized paths, eliminating the risk of unauthorized file uploads.

3. Improved Data Integrity: The combination of validation and sanitization measures significantly enhances data integrity by preventing malicious file uploads and protecting sensitive data.

While Scenario D provides robust security, it still has a potential limitation, storing the full filename in the database can lead to duplicate filenames and inconsistent naming conventions, potentially complicating database maintenance and monitoring. This could hinder efficient data management and retrieval.

Scenario E: Extension Separation and Attainment

Scenario E addresses the database storage limitation of Scenario D by implementing extension separation and storage:

1. Extension Separation: File extensions are separated from file names during processing. This helps prevent duplicate filenames and inconsistencies in naming conventions.
2. Extension Storage as Attribute: The extracted file extensions are stored as separate attributes in the database. This facilitates efficient data management and retrieval based on file extensions.

Scenario F: Enhanced Security Features

Scenario F builds upon Scenario E by incorporating additional security features, further strengthening the application's defense against file upload attacks:

1. Custom Hash-Salt Generation: A custom hash-salt is generated for each uploaded file. This unique identifier helps distinguish legitimate files from malicious ones and facilitates file verification.
2. Metadata Removal and Resizing: For image files, a photo-shoot mode is implemented to remove metadata before resizing. This helps eliminate potential hidden information that could be exploited for malicious purposes.
3. Responsive Design Resizing: Images are resized according to target media requirements using responsive design principles. This optimizes image display for different devices while reducing storage space.
4. Unique Filename Generation: A unique filename is generated using a combination of storage location, custom hash-salt, resizing format, and extension. This ensures consistent and identifiable filenames.
5. Original File Deletion: The original uploaded file is deleted from the server after resizing and metadata removal. This prevents unnecessary storage of potentially vulnerable files.

4.2. Summary of Findings

The research conducted on file upload security in web applications has identified several critical vulnerabilities and proposed a robust security approach based on the findings from various scenarios. The proposed approach builds upon existing research for generating unique file identifiers, offering an alternative to the hash-encryption model. It introduces innovative security measures for file storage in both storage media and databases.

Furthermore, the study adheres to and implements the Secure SDLC practices recommended by OWASP Top Ten Web Security (Casola et al., 2024; Jakimoski et al., 2022; Lala et al., 2021; Sharif, 2022). The key takeaways are:

1. Secure SDLC Implementation: The importance of adopting Secure SDLC practices throughout the development lifecycle is emphasized to ensure the integration of security measures from the early stages.
2. Layered Security Approach: A layered approach to file upload security is essential, incorporating both client-side and server-side validation, sanitization, and access control mechanisms.
3. Scenario F as Best Practice: Scenario F, which incorporates enhanced security features such as custom hash-salt generation, metadata removal, responsive design resizing, unique filename generation, and original file deletion, is recommended as the best practice for file upload functionalities.
4. Innovation in File Storage Security: The research introduces innovative approaches to secure file storage, including extension separation and storage, and secure file deletion practices.
5. Need for Further Research: The study highlights the need for further research in file validation and verification, resilience testing, forensic investigations, and attack pattern mapping in the context of file upload security.

4.3. Analysis and Recommendations

The research findings provide valuable insights into securing file uploads in web applications. The proposed scenario F offers a comprehensive and effective approach to mitigate file upload-related security risks. However, the study also acknowledges the need for continuous improvement and further research in this area.

4.4. Conclusion

The research on file upload security in web applications has made significant contributions to the field by highlighting the importance of Secure SDLC, layered security approaches, and innovative security features. The proposed scenario F serves as a valuable reference for implementing secure file upload functionalities. Further research in resilience testing, forensic investigations, attack pattern mapping, and validation and verification refinement is crucial for continuous improvement and adaptation to evolving threats.

4.5. Recommendations for Secure File Upload

Based on the findings, the following recommendations are proposed for securing file uploads in web applications:

1. Adopt Secure SDLC Practices: Integrate security considerations throughout the development lifecycle, from planning and design to implementation and testing.
2. Layered Validation and Sanitization: Employ multiple layers of validation and sanitization, both on the client-side and server-side, to protect against diverse attack vectors.

3. Strict Access Control: Implement strict access control mechanisms for storage directories, preventing unauthorized access and potential vulnerabilities.
4. Data Integrity Protection: Prioritize data integrity by ensuring that uploaded files are genuine, unmodified, and free from malicious code or data corruption.
5. Database Storage Optimization: Consider optimizing database storage of file information to avoid duplicate filenames and maintain consistent naming conventions.

By implementing these recommendations, web application developers can create secure and reliable file upload functionalities that safeguard sensitive data and protect users from potential security threats. Conclusions should clearly indicate the results obtained, their advantages and disadvantages, and possible future developments.

The conclusion can be a paragraph, but it should be point-to-point using numbering.

5. SUGGESTED

To address emerging challenges and further strengthen file upload security, the following areas are identified for future research:

1. Resilience Testing: Conduct resilience testing to evaluate the application's ability to withstand and recover from file upload attacks.
2. Forensic Investigations: Develop forensic investigation techniques for analyzing file upload-related security incidents.
3. Attack Pattern Mapping: Identify and map common attack patterns associated with file upload vulnerabilities.
4. Validation and Verification Refinement: Explore advanced validation and verification techniques to enhance file upload security.
5. Continuous Monitoring and Improvement: Establish continuous monitoring and improvement practices to address emerging threats and vulnerabilities.

6. REFERENCES

[1] Alghawazi, M., Alghazzawi, D., & Alarifi, S., 2022, Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review. Journal of Cybersecurity and Privacy, 2(4), 764–777.

[2] Al-Khannak, R., & Nehal, S. S., 2023, Penetration Testing for the Cloud-Based Web Application. WSEAS TRANSACTIONS ON COMPUTERS, 22, 104–113.

[3] Chen, Y., Li, Y., Pan, Z., Lu, Y., Chen, J., & Ji, S., 2024, URadar: Discovering Unrestricted File Upload Vulnerabilities via Adaptive Dynamic Testing. IEEE Transactions on Information Forensics and Security, 19, 1251–1266.

[4] Mathew, A., 2021, Obfuscation Techniques for Magecart Detection and Prevention. International Journal of Computer Science and Mobile Computing, 10(2), 39–44.

[5] Hilmi, M. A. Al, & Yunan, R. K., 2022, Pengujian Keamanan Fitur Upload File Pada Sistem Aplikasi Web. Jurnal Informatika: Jurnal Pengembangan IT, 7(1), 37–42

[6] Sharif, M. H., 2022, Web Attacks Analysis and Mitigation Techniques. International Journal of Engineering Research & Technology (IJERT)

[7] Upadhyay, D., & Ware, N. R., 2023, Evolving Trends in Web Application Vulnerabilities: A Comparative Study of OWASP Top 10 2017 and OWASP Top 10 2021 Article in. International Journal of Engineering Technology and Management Sciences.

[8] Lala, S. K., Kumar, A., & Subbulakshmi, T., 2021, Secure web development using OWASP guidelines. Proceedings - 5th International Conference on Intelligent Computing and Control Systems, ICICCS 2021, 323–332.

[9] Chawda, M., Sharma, Dr. P., & Patel, Mr. J., 2021, Deep Dive into Directory Traversal and File Inclusion Attacks leads to Privilege Escalation. International Journal of Scientific Research in Science, Engineering and Technology, 115–120.

[10] David Odera, Martin Otieno, & Jairus Ekume Ounza., 2023,. Security risks in the software development lifecycle: A review. World Journal of Advanced Engineering Technology and Sciences, 8(2), 230–253.

[11] Ibrahim, A., El-Ramly, M., & Badr, A., 2019, Beware of the Vulnerability! How Vulnerable are GitHub's Most Popular PHP Applications? 16th ACS/IEEE International Conference on Computer Systems and Applications AICCSA 2019 : 3 November to 7 November 2019, Al Ain University & Crowne Plaza, Abu Dhabi, UAE.

[12] Kumar, S. A., & Rani, Y. U., 2022, Implementation and analysis of Web application security measures using OWASP Guidelines. 2022 International Conference on Recent Trends in Microelectronics, Automation, Computing and Communications Systems (ICMACC), 182–187.

[13] R. Fujdiak et al., 2019, Managing the Secure Software Development : 24-26 June 2019, Canary Islands - Spain. 2019 9th IFIP International Conference on New Technologies, Mobility & Security : Proceedings of NTMS 2019 Conference and Workshop, 1–4.

[14] Chai, K. Y., & Zolkipli, M. F., 2021, Review on Confidentiality, Integrity and Availability in Information Security. Journal of ICT In Education, 8(2), 34–42.

[15] Casola, V., De Benedictis, A., Mazzocca, C., & Orbinato, V., 2024,. Secure software development and testing: A model-based methodology. Computers and Security, 137.

[16] Jakimoski, K., Stefanovska, Z., & Stefanovski, V., 2022, Optimization of Secure Coding Practices in SDLC as Part of Cybersecurity Framework. Journal of Computer Science Research, 4(2), 31–41.