Sri Lanka Institute of Information Technology

Specialized in Cyber Security

Year 2, Semester 2

Weekend Group

IE2062 - Web Security

The Bug Bounty Reports

IT21184758 – Liyanage P.P.

Table of Contents

Reports

Report 1

Vulnerability Title:

Path Traversal

Vulnerability Description:

An attacker might access sensitive files from the server and evade access controls by using the application's vulnerability to a path-traversal attack. This happens when processing user-supplied input for file operations because input validation and sanitization are insufficient.

Affected Components:

The path traversal vulnerability compromises the application's file processing capabilities. It specifically affects the systems for downloading and retrieving files.

Impact Assessment:

Because this flaw enables an attacker to access files that are supposed to be protected, it poses a serious danger. The impact can involve the disclosure of sensitive data, including user passwords, configuration files, or other private information, depending on the context and permissions of the program. Additionally, if the attacker succeeds in exploiting the obtained files, it can result in remote code execution.

Steps to Reproduce:

1. Open the program, then select the option for downloading files.

2. Use a proxy tool, such as Burp Suite, to intercept the request to download the file.

3. Add "../" or other directory traversal sequences to the file argument in the request.

4. Send the modified request, then monitor the outcome.

5. The vulnerability is verified if the server provides the contents of a file that isn't in the specified directory.

Proof of Concept:

Consider the following request to download a file:

```
GET /download?file=../config/users.txt HTTP/1.1

Host: business.kayak.com
```

In this example, the attacker modifies the "file" parameter to traverse outside the expected directory structure and retrieve the sensitive "user.txt" file.

Proposed Mitigation or Fix:

The following actions should be taken to lessen this vulnerability:

1. Input validation and sanitization: To prevent directory traversal sequences from being interpreted, make sure that all user-supplied input, especially parameters relating to files, undergoes extensive validation and sanitization.

2. Whitelist file access: Put in place a stringent file access control system that expressly permits access to only permitted files and folders.

3. Use secure file handling APIs: Utilize secure file handling routines or libraries that automatically handle path traversal and guard against unwanted file access.

4. Regular security assessments: Conduct routine security audits, including penetration testing and vulnerability scanning, to find and fix route traversal problems.

5. Stay up-to-date: Keep the program and all of its dependencies patched and updated to fix any known security flaws or vulnerabilities.

Report 2

Vulnerability Title:

SQL Injection

Vulnerability Description:

The program is vulnerable to SQL injection attacks, which allow a perpetrator to alter database queries by using erroneous user input. This flaw results from the incorrect sanitization and validation of user-provided data before it is used in SQL queries.

Affected Components:

Any feature or component that communicates with a backend database is vulnerable to SQL injection. This covers user authentication, data retrieval, search capabilities, and any other database-related functionalities.

Impact Assessment:

Because it enables an attacker to run arbitrary SQL statements inside the application's database, this vulnerability poses a serious danger. The possible effects include elevated privileges, illegal access to sensitive data, alteration or deletion of data, and potentially remote code execution if more vulnerabilities are found.

Steps to Reproduce:

1. Recognize a user input field that is used by an application's SQL query.

2. Insert a malicious SQL query or payload into the input field. As an instance:

    'OR '1'='1'; --

3. Submit the updated data and watch how the program responds.

4. The SQL injection vulnerability is validated if the program behaves unexpectedly, such as by displaying unwanted data or producing error messages that include database-related information.

Proof of Concept:

The application uses the provided credentials to accomplish the user login in the example below:

```
POST /login HTTP/1.1

Host: business.kayak.com

Content-Type: application/x-www-form-urlencoded


username=admin' OR '1'='1'; --

password=password123
```

In this scenario, the attacker injects a SQL payload into the username column, causing the query to always return true and thereby bypassing authentication.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Prepared statements or parameterized queries: Use parameterized queries or prepared statements with placeholders for user input. This guarantees that input given by the user is regarded as data rather than executable code, essentially avoiding SQL injection attacks.

2. Input validation and sanitization: Use tight input validation to reject or sanitize user input to avoid SQL metacharacters and escape sequences from being injected.

3. Principle of least privilege: Implement the concept of least privilege by assigning just the essential database privileges to the application's database user account, hence reducing the potential impact of successful SQL injection attacks.

4. Web application firewall (WAF): Use a WAF with SQL injection detection and prevention capabilities to give an extra layer of protection against such attacks.

5. Regular security assessments: Routine security assessments, including as penetration testing and code reviews, should be performed to discover and mitigate SQL injection issues.

6. Secure coding practices: Develop safe coding standards for developers, emphasizing the significance of validating and sanitizing user input and conducting regular code reviews to discover and repair any vulnerabilities.

Report 3

Vulnerability Title:

Time-Based SQL Injection allows unauthorized database access (Oracle)

Vulnerability Description:

The program is vulnerable to temporal-Based SQL Injection, which allows an attacker to alter SQL queries by taking advantage of temporal delays. This flaw originates from insufficient input validation and sanitization when combining user-supplied data into SQL queries run by an Oracle database.

Affected Components:

Any component or activity that interacts with an Oracle database is vulnerable to the Time-Based SQL Injection vulnerability. This comprises user authentication, data retrieval, search operations, and any other database-related functionality.

Impact Assessment:

This vulnerability is serious since it allows an attacker to extract critical information from a database, modify data, and potentially obtain unauthorized access. An attacker can circumvent security protections, exfiltrate data, and further abuse the database by carefully crafting payloads that cause latency delays in SQL queries.

Steps to Reproduce:

1. Locate a user input field that is used in a SQL query run by the Oracle database.

2. Place a timed SQL payload in the input field.

```
' AND DBMS_LOCK.SLEEP(5); --
```

3. Pay attention to the application's answer. The vulnerability is validated if the program encounters a considerable delay or shows behavior indicating a successful time-based SQL injection.

Proof of Concept:

Consider the following scenario, in which the program does a search based on user input:

```
POST /search HTTP/1.1

Host: business.kayak.com

Content-Type: application/x-www-form-urlencoded


search_term=widget' AND DBMS_LOCK.SLEEP(5); --
```

In this scenario, the attacker injects a time-delayed SQL payload into the search_term field, forcing the query to stall for 5 seconds, indicating that the Time-Based SQL Injection was successful.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Input validation and sanitization: To avoid the insertion of malicious SQL payloads, provide effective input validation and sanitization algorithms. To segregate user input from SQL queries, utilize parameter binding or bind variables.

2. Least privilege principle: Assign database accounts using the least privilege principle, allowing just the rights required for the application to work effectively. Limit the app's access to sensitive data or actions.

3. Web application firewall (WAF): To filter out potentially dangerous requests, deploy a WAF with time-based SQL injection detection and prevention capabilities.

4. Patching and updates: To address any known vulnerabilities, keep the Oracle database and accompanying software up to date with the most recent security patches and updates.

5. Regular security assessments: Conduct frequent security audits, including penetration testing, to discover and address Time-Based SQL Injection issues.

6. Security awareness and training: Educate developers on secure coding methods, emphasizing the need of input validation, parameter binding, and SQL injection prevention.

Report 4

Vulnerability Title:

SQL Injection allows unauthorized database access (SQLite)

Vulnerability Description:

SQL injection attacks are possible against the program, allowing an attacker to modify database queries using unvalidated user input. This vulnerability exists because user-supplied data is not properly sanitized and validated before being incorporated into SQL queries performed by a SQLite database.

Affected Components:

Any component or activity that interacts with a SQLite database is vulnerable to SQL injection. This comprises user authentication, data retrieval, search operations, and any other database-related functionality.

Impact Assessment:

This vulnerability is serious since it allows an attacker to run arbitrary SQL statements on the application's SQLite database. If further vulnerabilities are uncovered, the possible impact includes unauthorized access to sensitive data, data modification or deletion, privilege escalation, and potentially remote code execution.

Steps to Reproduce:

1. Locate a user input field that is used in a SQL query run by the SQLite database.

2. Insert a rogue SQL query or payload into the input field.

As an example: ' OR 1=1; --

3. Submit the changed input and wait for the application's response.

4. The SQL injection vulnerability is validated if the program behaves unexpectedly, such as displaying unwanted data or creating error messages disclosing database-related information.

Proof of Concept:

Consider the following scenario, in which the program conducts a user login based on the credentials provided:

```
POST /login HTTP/1.1

Host: business.kayak.com

Content-Type: application/x-www-form-urlencoded



username=admin' OR 1=1; --

password=password123
```

In this scenario, the attacker injects a SQL payload into the username column, causing the query to always return true and thereby circumventing authentication.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Prepared statements or parameterized queries: Use parameterized queries or prepared statements with placeholders for user input. This guarantees that input given by the user is regarded as data rather than executable code, essentially avoiding SQL injection attacks.

2. Input validation and sanitization: Use stringent input validation to reject or sanitize user input to avoid SQL metacharacters and escape sequences from being injected.

3. Secure coding techniques: Educate developers on secure coding standards, emphasizing the necessity of validating and sanitizing user input as well as conducting frequent code reviews to discover and repair any vulnerabilities.

4. Regular security assessments: Routine security assessments, including as penetration testing and code reviews, should be performed to discover and mitigate SQL injection issues.

5. Stay up-to-date: Keep the SQLite library and accompanying software components up to date with the most recent security patches and upgrades to address any known vulnerabilities.

6. Principle of least privilege: Assign database accounts using the least privilege approach, allowing just the rights required for the application to work effectively. Limit the app's access to sensitive data or actions.

Report 5

Vulnerability Title:

Wildcard Directive in Content Security Policy allows unsafe content loading

Vulnerability Description:

The application's material Security Policy (CSP) has a wildcard directive that permits dangerous material from any source to be loaded. This misconfiguration may provide security issues since it circumvents the CSP's intended constraints, allowing the execution of malicious scripts and the loading of untrusted resources.

Affected Components:

The Content Security Policy's wildcard directive impacts the whole application, affecting all pages and resources that are subject to the CSP.

Impact Assessment:

The wildcard directive in the material Security Policy provides a substantial risk by enabling dangerous material from any source to be loaded. This might result in a variety of security risks, including cross-site scripting (XSS) attacks, unauthorized data exfiltration, and client-side vulnerability exploitation. It compromises the CSP's protection and may expose users to malicious actions.

Steps to Reproduce:

1. Navigate to the application and look for the Content Security Policy (CSP) header in the server response.

2. Determine whether the CSP has a wildcard (*) directive.

3. Load a website or resource that violates the CSP constraints, such as an external or inline script.

4. Check to see if the content is loaded and performed without any CSP warnings or limitations.

5. If the content is successfully loaded and processed, the vulnerability associated with the wildcard directive is validated.

Proof of Concept:

For instance, suppose the application's CSP header includes the following directive:

```
Content-Security-Policy: default-src 'self' *;
```

The default-src policy's wildcard (*) directive allows the material to be loaded from any source, weakening the intended security limitations.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Restrictive CSP policies: Examine and amend the CSP to incorporate specific directives that restrict content loading to just trustworthy sources. Unless absolutely required, utilize wildcard directives (*).

2. Whitelisting trusted sources: Specify trusted sources explicitly using suitable CSP directives such as'self', 'https:', or particular domain names.

3. Strict mode and nonce values: Use strict-dynamic mode or nonce values to regulate script loading and execution, allowing only trusted scripts to run.

4. Regular CSP header audits: Audit the CSP headers on a regular basis to ensure their efficacy and to discover and correct any misconfigurations or vulnerabilities.

5. Content Security Policy reporting: Enable reporting methods to collect and evaluate violation reports, enabling for proactive detection of potential CSP bypasses or attacks.

6. Secure coding procedures: Use secure coding methods during the development phase to avoid the introduction of dangerous content-loading vulnerabilities.

Report 6

Vulnerability Title:

Unsafe Inline Script Directive in Content Security Policy allows execution of arbitrary scripts

Vulnerability Description:

The "unsafe-inline" directive is included in the "script-src" policy of the application's Content Security Policy (CSP). This directive permits the execution of arbitrary inline scripts, circumventing the CSP's intended constraints. It makes the application vulnerable to cross-site scripting (XSS) attacks and other script-based flaws.

Affected Components:

The "unsafe-inline" directive in the "script-src" policy affects all pages and resources in the application that rely on inline script execution.

Impact Assessment:

The "unsafe-inline" directive's presence in the "script-src" policy constitutes a severe security concern. It permits arbitrary scripts, including possibly dangerous code, to be executed within the program. This can result in XSS attacks, unauthorized data access or alteration, and the exploitation of client-side vulnerabilities.

Steps to Reproduce:

1. Open the application and examine the server response for the Content Security Policy (CSP) header.

2. Examine the "script-src" policy directive for the presence of the "unsafe-inline" directive.

3. Open an application page that has an inline script.

4. Ensure that the inline script runs without any CSP warnings or limitations.

5. The vulnerability associated with the "unsafe-inline" directive is proven if the inline script executes properly, confirming the bypass of the CSP.

Proof of Concept:

For instance, suppose the application's CSP header includes the following directive:

```
Content-Security-Policy: script-src 'self' 'unsafe-inline';
```

The presence of the "unsafe-inline" directive permits the execution of inline scripts even if they break the CSP constraints.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Strict CSP regulations: Remove the "unsafe-inline" directive from the "script-src" policy and implement tighter script execution restrictions.

2. Externalize scripts: Refactor the program so that inline scripts are separated and loaded as external files from trustworthy sources.

3. Hash- or nonce-based execution: Use script integrity hashes or nonces to verify that only trusted scripts are executed.

4. Content Security Policy reporting: Allow reporting methods to receive and evaluate violation reports, enabling for proactive detection of inline script use and related vulnerabilities.

5 Secure coding techniques: Educate developers about the hazards of inline scripts and encourage them to utilize best practices like externalizing scripts and avoiding inline code.

6. Regular CSP header audits: Audit the CSP headers on a regular basis to ensure their efficacy and to discover and correct any misconfigurations or vulnerabilities.

Report 7

Vulnerability Title:

Unsafe Inline Style Directive in Content Security Policy allows execution of arbitrary styles

Vulnerability Description:

The "unsafe-inline" directive is included in the "style-src" policy of the application's Content Security Policy (CSP). This directive lets the application to use inline styles while avoiding the CSP's intended constraints. It makes the application vulnerable to style-based vulnerabilities and raises the likelihood of cross-site scripting (XSS) attacks.

Affected Components:

The "unsafe-inline" directive in the "style-src" policy impacts all pages and resources in the application that rely on inline style execution.

Impact Assessment:

The "unsafe-inline" directive's presence in the "style-src" policy provides a severe security concern. It permits the program to execute arbitrary styles, including possibly harmful code. This can result in XSS attacks, unauthorized data access or alteration, and the exploitation of client-side vulnerabilities.

Steps to Reproduce:

1. Open the application and examine the server response for the Content Security Policy (CSP) header.

2. Examine the "style-src" policy directive to see whether it contains the "unsafe-inline" directive.

3. Open an application page that contains an inline style.

4. Check that no CSP-related warnings or limitations are present when the inline style is applied.

5. The vulnerability related with the "unsafe-inline" directive is proven if the inline style is successfully implemented, confirming the bypass of the CSP.

Proof of Concept:

For instance, suppose the application's CSP header includes the following directive:

```
Content-Security-Policy: style-src 'self' 'unsafe-inline';
```

The introduction of the "unsafe-inline" directive allows inline styles to be executed even if they break CSP limitations.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Strict CSP policies: Remove the "unsafe-inline" directive from the "style-src" policy and impose stricter style application restrictions.

2. Externalize styles: Refactor the program so that inline styles are separated and applied via external CSS files fetched from trusted sources.

3. Nonce-based or hash-based execution: Use style integrity hashes or nonces to verify that only trustworthy styles are used.

4. Content Security Policy reporting: Allow reporting systems to receive and evaluate violation reports, enabling for proactive detection of inline style use and possible vulnerabilities.

5. Secure coding methods: Educate developers on the dangers of inline styles and encourage them to utilize best practices such as externalizing styles and avoiding inline code.

6. Regular CSP header audits: Audit the CSP headers on a regular basis to ensure their efficacy and to discover and correct any misconfigurations or vulnerabilities.

Report 8

Vulnerability Title:

Missing Content Security Policy (CSP) Header allows the potential for various security risks

Vulnerability Description:

In the server response, the application does not provide a Content Security Policy (CSP) header. Because it lacks the essential safeguards to minimize cross-site scripting (XSS), data injection, and other client-side vulnerabilities, this absence exposes the program to possible security concerns. Without a CSP, the application becomes more vulnerable to attacks that take advantage of illegal script execution and the loading of untrusted resources.

Affected Components:

Because it applies to all pages and resources delivered by the application, the absence of a Content Security Policy (CSP) header has an impact on the entire program.

Impact Assessment:

The absence of a Content Security Policy (CSP) header makes the application more vulnerable to different client-side attacks. It enables for possible XSS attacks, illegal data exfiltration, and client-side vulnerability exploitation. Furthermore, the lack of a CSP hinders the application from imposing the required constraints on script execution, resource loading, and inline style usage, exposing users to a higher level of risk.

Steps to Reproduce:

1. Navigate to the application and inspect the server response headers.

2. Look for a missing Content Security Policy (CSP) header.

3. Open an application page that has user-controllable inputs or dynamic content.

4. Check to see if the website runs scripts or loads resources from untrusted sources without any CSP warnings or limitations.

5. The vulnerability associated with the missing CSP header is proven if the website supports the execution of unauthorized scripts or the loading of untrusted resources.

Proof of Concept:

Analyze the server response headers to ensure the absence of the Content Security Policy (CSP) header. Search for the following heading:

Content-Security-Policy:

If the header is absent, it indicates that the CSP header is missing.

Proposed Mitigation or Fix:

To address this issue, the following steps should be taken:

1. Put in place a Content Security Policy (CSP): For all pages and resources delivered by the application, include an appropriate CSP header in the server response.

2. Establish restrictive policies: Configure the CSP with policies that restrict script execution, resource loading, and the use of inline styles to trusted sources only.

3. Whitelist reliable sources: Using the relevant CSP directives, such as'self,' 'https:,' or particular domain names, explicitly designate trustworthy sources.

4. CSP header audits on a regular basis: Conduct frequent audits to ensure that the CSP header is there and active, and that it is accurately configured and actively enforced.

5. Content Security Policy reporting: Enable reporting methods to collect and evaluate violation reports, enabling for proactive detection of potential CSP bypasses or assaults.

6. Secure coding practices: Educate developers on the importance of implementing CSP headers and the risks associated with client-side vulnerabilities. Encourage the use of secure coding practices to prevent the introduction of security vulnerabilities during development.

Report 9

Vulnerability Title:

Cross-Domain Misconfiguration allows unauthorized access to sensitive resources

Vulnerability Description:

The cross-domain setup of the application is incorrect, allowing unauthorized access to critical resources across domains. This misconfiguration can lead to data leakage, unauthorized data alteration, and the exploitation of critical functions.

Affected Components:

Misconfigured cross-domain settings have an influence on the application's communication with other domains or subdomains, potentially disrupting numerous resources and functions that rely on cross-domain interactions.

Impact Assessment:

Misconfiguration of cross-domain settings poses security concerns with serious repercussions. It enables attackers to circumvent specified security measures and get access to sensitive resources or functionality housed across domains. The effect includes unauthorized data access, alteration, or exfiltration, as well as the possibility of privilege escalation and cross-domain vulnerability exploitation.

Steps to Reproduce:

1. Determine the domains and subdomains of the application that are involved in cross-domain communication.

2. Examine the cross-domain setup, including CORS headers, iframe settings, and any other methods engaged in cross-domain interactions.

3. Unauthorized access to critical resources or functionality from a different domain or subdomain.

4. Check to see if the application permits access to sensitive resources or capabilities without any cross-domain limitations or warnings.

5. If unauthorized access is successful, the vulnerability associated with the misconfiguration is confirmed.

Proof of Concept:

For example, if the application's cross-domain setup lacks suitable CORS headers or permits unfettered cross-domain communication, an attacker might possibly access sensitive resources or execute illegal actions.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Examine and update cross-domain configuration: Examine the cross-domain communication methods of the application, including CORS headers, iframe settings, and any other relevant setups. Check that they are correctly designed and restricted.

2. Correctly implement CORS: To prevent cross-domain access to critical sites, provide suitable CORS headers in answers. Using the "Access-Control-Allow-Origin" header, only allow access from trustworthy domains.

3. Limit cross-domain access: Consider imposing stricter validation for cross-domain requests and creating more specific cross-domain regulations, such as identifying permissible methods and headers.

4. Audits on a regular basis: Conduct frequent cross-domain configuration audits to detect and correct any misconfigurations or vulnerabilities. Check that the settings are in accordance with the application's security needs.

5. Secure code procedures: Educate developers on the dangers associated with cross-domain misconfigurations and advocate secure coding standards that prevent such vulnerabilities from being introduced throughout the development process.

6. Security testing: Conduct thorough security testing, such as penetration testing and vulnerability scanning, to uncover potential cross-domain misconfigurations and other security concerns.

Report 10

Vulnerability Title:

Hidden File Disclosure allows unauthorized access to sensitive information

Vulnerability Description:

The program reveals hidden files or folders containing sensitive data. These secret files are designed to be unavailable to users and should not be made public. Hidden files raise the danger of unwanted access to critical data, potentially resulting in data leaks and security breaches.

Affected Components:

The hidden file disclosure vulnerability affects certain files or directories that have been inadvertently disclosed and made available to unauthorized users.

Impact Assessment:

The disclosure of secret data poses a serious security risk. It might lead to unauthorized access to sensitive data such as configuration files, source code, database backups, credentials, or other secret information. This information can be used to acquire further system access, initiate attacks, or damage the application's security and integrity.

Steps to Reproduce:

1. Determine whether any hidden files or folders exist within the application.

2. Examine the application's file or directory structure for hidden files.

3. Attempt to directly access the hidden files or folders using proper URLs or directory traversal techniques.

4. Check to see if the hidden files can be accessed without authentication or sufficient authorisation.

5. The issue is proven if unauthorized access to hidden files is successful, indicating the vulnerability associated with hidden file disclosure.

Proof of Concept:

Accessing a hidden file named "._darcs" or a secret directory named ".hg" that includes sensitive information, for example, would indicate the presence of a hidden file disclosure vulnerability.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Examine and protect hidden files: Locate and examine all hidden files or folders within the application. Remove any critical information from these files and assure their security.

2. Restrict access to hidden files: To prevent unwanted access to hidden files, use access restrictions such as proper file permissions or authentication systems.

3. Server configuration: Set up the web server to prevent hidden files or folders from being exposed. Modifying server settings, such as stopping directory indexing or creating proper file access restrictions, may be required.

4. Security audits on a regular basis: Conduct security audits on a regular basis to detect and resolve any exposed or vulnerable hidden files. Implement safeguards to avoid the unintentional disclosure of sensitive information.

5. Secure coding techniques: Educate developers on the necessity of properly safeguarding sensitive information and employing secure coding methods to avoid hidden file disclosure vulnerabilities from being introduced during development.

6. Penetration testing: Conduct extensive penetration testing to uncover hidden file disclosure vulnerabilities and other application security flaws.

Report 11

Vulnerability Title:

Missing Anti-Clickjacking Header allows clickjacking attacks

Vulnerability Description:

In the server answers, the application does not provide an Anti-Clickjacking header, such as X-Frame-Options or Content-Security-Policy frame-ancestors directive. This omission exposes the program to clickjacking attacks, in which an attacker may deceive users into doing unwanted actions by overlaying malicious information or frames on top of the application.

Affected Components:

The absence of an Anti-Clickjacking header impacts all sites and resources delivered by the application, making them vulnerable to clickjacking attacks.

Impact Assessment:

The lack of an Anti-Clickjacking header increases the possibility of clickjacking assaults, which can have a variety of consequences, including illegal activities conducted by users who inadvertently engage with harmful material. This might result in unintentional changes to settings, data manipulation, unlawful transactions, or sensitive information theft.

Steps to Reproduce:

1. Navigate to the application and examine the server response headers.

2. Examine the Content-Security-Policy header for the lack of an Anti-Clickjacking header, such as X-Frame-Options or frame-ancestors directive.

3. Create a page or utilize an already existing malicious website to embed the application's content in an iframe.

4. Open the crafted page or the malicious website with the embedded program.

5. Check to see if the program has been correctly integrated within the iframe without any clickjacking protection.

6. The vulnerability associated with missing protection is proven if the program is embedded without protection, indicating the lack of an Anti-Clickjacking header.

Proof of Concept:

If the application's server responses lack an X-Frame-Options header with the "SAMEORIGIN" or "DENY" value, or if the Content-Security-Policy header lacks a frame-ancestors directive restricting embedding to trusted sources, it confirms the absence of Anti-Clickjacking protection.

Proposed Mitigation or Fix:

The following steps should be made to mitigate this vulnerability:

1. Use Anti-Clickjacking headers: In the server answers for all pages and resources delivered by the application, include an appropriate Anti-Clickjacking header, such as X-Frame-Options or Content-Security-Policy frame-ancestors directive.

2. X-Frame-Options header: Set the "SAMEORIGIN" value in the X-Frame-Options header to allow embedding only from the same origin, or use "DENY" to disable all framing.

3. Content-Security-Policy header: Use the frame-ancestors directive to restrict embedding to trustworthy sources in the Content-Security-Policy header.

4. Conduct regular header audits: to ensure that the Anti-Clickjacking headers are accurately established and aggressively enforced.

5. Security awareness: Inform developers about the dangers of clickjacking attacks and the need of using Anti-Clickjacking headers. Encourage the usage of safe coding methods throughout development to avoid the introduction of clickjacking vulnerabilities.