

[Scan Results for vuln.stенаeке.оrg](#)  
[Scanning in progress](#)  
[Absence of Anti-CSRF Tokens](#)  
[Content Security Policy \(CSP\) Header Not Set](#)  
[Missing Anti-clickjacking Header](#)  
[Cookie without SameSite Attribute](#)  
[X-Content-Type-Options Header Missing](#)  
[Authentication Request Identified](#)  
[Retrieved from Cache](#)  
[Session Management Response Identified](#)  
[User Agent Fuzzer](#)  
[Cookie Without Secure Flag](#)  
[Cookie without SameSite Attribute](#)  
[Strict-Transport-Security Header Not Set](#)

# Scan Results for vuln.stенаeке.оrg

Scan started: 2025-04-10 10:30:41

Initializing scan... Please wait while we analyze the target.

## Scanning in progress

The scan is now running. Results will appear here as they are processed.

## Absence of Anti-CSRF Tokens

### Issue Explanation

The absence of anti-CSRF tokens in HTML submission forms can be a significant security vulnerability. CSRF (Cross-Site Request Forgery) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. This is done by tricking the user into submitting a request to the application, which the application then processes as if it were from the user.

In the context of HTTP headers, CSRF attacks can be executed by an attacker embedding a malicious link or form in a website that the victim visits. When the victim clicks the link or submits the form, the browser sends a request to the target application, which the application processes as if it were from the victim.

Without anti-CSRF tokens, the application cannot differentiate between legitimate requests

from the victim and malicious requests from an attacker. This can lead to unauthorized actions being performed on behalf of the victim, such as changing account information, transferring funds, or executing other sensitive operations.

## Impact Analysis

The impact of CSRF without anti-CSRF tokens can be severe, including:

1. **Unauthorized Actions:** Attackers can perform actions on behalf of the victim without their knowledge, leading to unauthorized changes to their account or data.
2. **Data Manipulation:** Attackers can manipulate data, such as changing account information, transferring funds, or executing other sensitive operations.
3. **Session Hijacking:** Attackers can hijack the victim's session by tricking them into submitting a malicious request, allowing the attacker to access the victim's account.
4. **Phishing Attacks:** Attackers can use CSRF to trick users into submitting sensitive information to a malicious site, which can then be used for further attacks.
5. **Data Breach:** If the application processes sensitive data, CSRF can be used to exfiltrate this data without the victim's knowledge.

## Exploitation Details

To exploit CSRF without anti-CSRF tokens, attackers can:

1. **Craft Malicious Links:** Attackers can create links that point to the target application, tricking the victim into clicking them.
2. **Embed Malicious Forms:** Attackers can embed forms in a website that the victim visits, tricking them into submitting the form.
3. **Use of Reflected XSS:** If the application is vulnerable to reflected XSS, attackers can inject CSRF tokens into the response, allowing them to bypass the CSRF protection.
4. **Automated Tools:** Tools like BeEF (Browser Exploitation Framework) can be used to automate the process of creating and delivering CSRF attacks.

## Step-by-Step Remediation

To remediate CSRF without anti-CSRF tokens, you can:

1. **Server-Side Validation:** Implement server-side validation to ensure that the request is coming from a legitimate source.
2. **Use of Anti-CSRF Tokens:** Generate and include anti-CSRF tokens in forms and links, and validate them on the server-side.
3. **Use of CSRF Protection Libraries:** Use libraries like OWASP CSRFGuard or the OWASP ESAPI to implement CSRF protection.
4. **Use of HTTPS:** Ensure that the application uses HTTPS to prevent attackers from intercepting and modifying requests.

5. **User Education:** Educate users about the risks of CSRF and how to recognize and avoid malicious links and forms.

## References & Best Practices

- [OWASP CSRF Prevention Cheat Sheet](#)
  - [OWASP CSRF Guard Project](#)
  - [OWASP ESAPI Project](#)
  - [OWASP CSRF Prevention Cheat Sheet](#)
  - [OWASP CSRF Guard Project](#)
  - [OWASP ESAPI Project](#)
- 

## Content Security Policy (CSP) Header Not Set

### Issue Explanation

The vulnerability in question is related to the absence of the Content Security Policy (CSP) header in the HTTP response. CSP is a security feature that helps protect against cross-site scripting (XSS) and other code injection attacks by specifying which dynamic resources are allowed to load on a page.

The CSP header is used to define a whitelist of trusted sources for various types of resources, such as JavaScript, CSS, images, and more. By specifying the allowed sources, the browser can enforce the policy and prevent the execution of malicious scripts or the loading of malicious resources.

### Impact Analysis

The absence of the CSP header can have the following security implications:

1. **Increased Risk of XSS Attacks:** Without a CSP, the browser has no way to enforce the security policy, making it easier for attackers to inject malicious scripts into the page.
2. **Lack of Protection Against Data Injection Attacks:** CSP helps prevent data injection attacks by specifying which sources are allowed to load data into the page.
3. **Potential for Phishing and Malware Distribution:** Attackers can use the lack of CSP to inject malicious scripts or resources that can be used for phishing or malware distribution.
4. **Lack of Granular Control Over Resources:** Without CSP, there is no way to control which resources are loaded on the page, leading to a less secure environment.

### Exploitation Details

Attackers can exploit the absence of CSP by:

1. **Injecting Malicious Scripts:** Attackers can inject malicious scripts into the page, which can be executed by the browser without any restrictions.

2. **Loading Malicious Resources:** Attackers can load malicious resources, such as images or iframes, which can be used to execute attacks or distribute malware.
3. **Phishing and Social Engineering:** Attackers can use the lack of CSP to craft phishing emails or social engineering attacks that appear more convincing.

## Step-by-Step Remediation

To remediate this vulnerability, you can set the CSP header on your web server or application server. Here are some examples for different platforms:

### 1. Apache HTTP Server:

- Add the following directive to your `.htaccess` file or `httpd.conf` file:

```
Header set Content-Security-Policy "default-src'self'; script-s
```

- Replace `'self'` with the appropriate sources you want to allow.

### 2. Nginx:

- Add the following directive to your `nginx.conf` file:

```
add_header Content-Security-Policy "default-src'self'; script-s
```

- Replace `'self'` with the appropriate sources you want to allow.

### 3. IIS:

- Add the following directive to your `web.config` file:

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="Content-Security-Policy" value="default-src'se
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

- Replace `'self'` with the appropriate sources you want to allow.

### 4. Express.js:

- Use the `helmet` middleware to set the CSP header:

```
const express = require('express');
const helmet = require('helmet');
const app = express();

app.use(helmet.contentSecurityPolicy({
  useDefaults: true,
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'"],
    styleSrc: ["'self'"]
  }
})
```

```
}}};
```

- Replace 'self' with the appropriate sources you want to allow.

## 5. PHP:

- Use the `header()` function to set the CSP header in your PHP code:

```
header("Content-Security-Policy: default-src'self'; script-src'
```

- Replace 'self' with the appropriate sources you want to allow.

## References & Best Practices

- [OWASP CSP Cheat Sheet](#)
  - [MDN CSP Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

## Missing Anti-clickjacking Header

### Issue Explanation

The vulnerability in question is related to the absence of anti-clickjacking headers in the HTTP response. Clickjacking is a type of attack where an attacker tricks a user into clicking on a malicious link or button that appears to be part of a legitimate website. This can lead to the user unknowingly performing actions on the attacker's behalf, such as logging into their account or downloading malware.

The HTTP headers `Content-Security-Policy` and `X-Frame-Options` are used to protect against clickjacking attacks. The `Content-Security-Policy` header allows you to specify which domains are allowed to load resources (like scripts, images, etc.) on your page, while the `X-Frame-Options` header controls whether the page can be displayed in a frame or not.

### Impact Analysis

The absence of these headers can have the following security implications:

1. **Clickjacking Attacks:** Attackers can embed the vulnerable page in an iframe on a malicious website and trick users into clicking on it, leading to unauthorized actions being performed on the user's behalf.
2. **Phishing and Social Engineering:** Attackers can use clickjacking to trick users into performing actions on malicious websites that appear to be legitimate, leading to data theft or account compromise.
3. **Cross-Site Scripting (XSS):** Attackers can use clickjacking to bypass XSS protections by

tricking users into executing malicious scripts.

4. **Session Hijacking:** Attackers can use clickjacking to hijack user sessions by tricking users into clicking on a malicious link that appears to be part of a legitimate website.

## Exploitation Details

Attackers can exploit this vulnerability by:

1. **Embedding the Target Page:** The attacker can embed the vulnerable page in an iframe on a malicious website. When a user visits the malicious website, they will see the vulnerable page in the iframe, and any actions they perform on the page will be performed on the attacker's behalf.
2. **Tricking Users:** The attacker can use social engineering techniques to trick users into clicking on the malicious link or button, leading to the execution of the attacker's code.
3. **Automated Attacks:** Tools like BeEF (Browser Exploitation Framework) can be used to automate the process of embedding the vulnerable page and tricking users into performing actions.

## Step-by-Step Remediation

To remediate this vulnerability, you can:

### 1. Server-Level Remediation:

- **Apache:** Set the X-Frame-Options header to SAMEORIGIN or DENY to prevent the page from being framed. For example:

```
Header set X-Frame-Options SAMEORIGIN
```

- **Nginx:** Set the X-Frame-Options header to SAMEORIGIN or DENY to prevent the page from being framed. For example:

```
add_header X-Frame-Options SAMEORIGIN;
```

- **IIS:** Set the X-Frame-Options header to SAMEORIGIN or DENY to prevent the page from being framed. For example:

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="X-Frame-Options" value="SAMEORIGIN" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

### 2. Application-Level Remediation:

- **Express.js:** Use the frameguard middleware to set the X-Frame-Options header. For example:

```
app.use(require('frameguard')());
```

- **PHP:** Use the header function to set the X-Frame-Options header. For example:

```
header('X-Frame-Options: SAMEORIGIN');
```

## References & Best Practices

- [OWASP Clickjacking Defense Cheat Sheet](#)
  - [X-Frame-Options Documentation](#)
  - [Content Security Policy Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

## Cookie without SameSite Attribute

### Issue Explanation

The vulnerability in question is related to the absence of the SameSite attribute in a cookie. The SameSite attribute is a browser-side policy that helps mitigate the risk of cross-site request forgery (CSRF) and cross-site scripting (XSS) attacks by controlling the contexts in which a cookie is sent.

Cookies without the SameSite attribute can be sent as a result of a 'cross-site' request, which can lead to security issues. For example, if an attacker tricks a user into visiting a malicious website, the attacker can potentially access the cookie and perform actions on behalf of the user without their consent.

### Impact Analysis

The impact of not setting the SameSite attribute can be significant:

1. **CSRF Attacks:** Attackers can perform CSRF attacks by tricking users into visiting a malicious website that sends a request to the server with the cookie, allowing the attacker to perform actions on behalf of the user.
2. **XSS Attacks:** Attackers can perform XSS attacks by injecting malicious scripts into the page, which can access the cookie and perform actions on behalf of the user.
3. **Session Hijacking:** Attackers can hijack user sessions by stealing the cookie and using it to impersonate the user.
4. **Data Breach:** Attackers can access sensitive data stored in the cookie, leading to a data breach.

### Exploitation Details

Attackers can exploit this vulnerability by:

1. **Phishing:** Sending phishing emails or links to users that trick them into visiting a malicious website.
2. **Cross-Site Scripting (XSS):** Injecting malicious scripts into the page that can access the cookie.
3. **Cross-Site Request Forgery (CSRF):** Tricking users into performing actions on behalf of the attacker.

## Step-by-Step Remediation

To remediate this vulnerability, you can set the SameSite attribute to 'lax' or 'strict' for all cookies. Here's how you can do it for different platforms:

### 1. Server-Level Remediation:

- **Apache:** Set the SameSite directive in the Set-Cookie header. For example:

```
Set-Cookie: mycookie=value; SameSite=Lax
```

- **Nginx:** Set the same\_site directive in the add\_header directive. For example:

```
add_header Set-Cookie mycookie=value; SameSite=Lax;
```

- **IIS:** Set the SameSite attribute in the Set-Cookie header. For example:

```
Set-Cookie: mycookie=value; SameSite=Lax
```

### 2. Application-Level Remediation:

- **Express.js:** Set the SameSite attribute in the res.cookie() method. For example:

```
res.cookie('mycookie', 'value', { sameSite: 'Lax' });
```

- **PHP:** Set the SameSite attribute in the setcookie() function. For example:

```
setcookie('mycookie', 'value', ['SameSite' => 'Lax']);
```

## References & Best Practices

- [OWASP Documentation on SameSite](#)
  - [W3C Documentation on SameSite](#)
  - [Apache HTTP Server Documentation on SameSite](#)
  - [Nginx Documentation on SameSite](#)
  - [IIS Documentation on SameSite](#)
  - [Express.js Documentation on SameSite](#)
  - [PHP Documentation on SameSite](#)
- 

## X-Content-Type-Options Header Missing

### Issue Explanation



The vulnerability in question is related to the absence of the X-Content-Type-Options header, which is a security feature that can prevent MIME-sniffing attacks. MIME-sniffing is a technique used by some web browsers to guess the content type of a response based on the content of the response body, which can lead to security issues.

The X-Content-Type-Options header is used to instruct the browser to not perform MIME-sniffing on the response body. When this header is set to 'nosniff', it tells the browser to use the declared content type in the Content-Type header, rather than guessing it based on the response body.

## Impact Analysis

The absence of the X-Content-Type-Options header can have the following security implications:

1. **MIME-Sniffing Attacks:** Attackers can exploit the absence of this header to perform MIME-sniffing attacks, where they guess the content type of the response body and serve malicious content that is interpreted as a different type, leading to security issues.
2. **Cross-Site Scripting (XSS):** If the content type is guessed incorrectly, it can lead to XSS attacks, where malicious scripts are executed in the context of the website.
3. **Phishing and Social Engineering:** Attackers can use the absence of the X-Content-Type-Options header to serve phishing content that appears as a different type, making it more convincing to the user.
4. **Data Leakage:** If the content type is guessed incorrectly, it can lead to data leakage, where sensitive information is exposed to the attacker.

## Exploitation Details

Attackers can exploit the absence of the X-Content-Type-Options header by:

1. **MIME-Sniffing:** Using tools like Burp Suite or OWASP ZAP to intercept and analyze the HTTP traffic, and then serving malicious content that is interpreted as a different type.
2. **Cross-Site Scripting (XSS):** Injecting malicious scripts into the response body and relying on the browser to interpret the content as a different type, leading to XSS.
3. **Phishing and Social Engineering:** Crafting phishing emails or social engineering attacks that serve content that appears as a different type, making it more convincing to the user.

## Step-by-Step Remediation

To remediate this vulnerability, you can take the following steps:

### 1. Server-Level Remediation:

- **Apache:** Set the X-Content-Type-Options header to 'nosniff' in the server configuration. For example:

```
Header set X-Content-Type-Options nosniff
```

- **Nginx:** Set the X-Content-Type-Options header to 'nosniff' in the server

configuration. For example:

```
add_header X-Content-Type-Options nosniff;
```

- **IIS:** Set the X-Content-Type-Options header to 'nosniff' in the server configuration. For example:

```
<system.webServer>
  <httpProtocol>
    <customHeaders>
      <add name="X-Content-Type-Options" value="nosniff" />
    </customHeaders>
  </httpProtocol>
</system.webServer>
```

## 2. Application-Level Remediation:

- **Express.js:** Set the X-Content-Type-Options header to 'nosniff' in the application code. For example:

```
app.use((req, res, next) => {
  res.setHeader('X-Content-Type-Options', 'nosniff');
  next();
});
```

- **PHP:** Set the X-Content-Type-Options header to 'nosniff' in the application code. For example:

```
header('X-Content-Type-Options: nosniff');
```

## References & Best Practices

- [OWASP ZAP Documentation](#)
  - [Burp Suite Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

# Authentication Request Identified

## Issue Explanation

The alert "Authentication Request Identified" is an informational alert generated by OWASP ZAP when it detects an authentication request. This alert is not a vulnerability but provides information about the authentication method used by the target application.

The alert includes a set of key-value pairs in the "Other Info" field, which identify the authentication method used in the request. This information can be useful for security professionals to understand the authentication mechanisms employed by the target

application.

## Impact Analysis

The impact of this alert is limited to providing information about the authentication method used by the target application. It does not represent a security vulnerability itself. However, this information can be useful for security professionals to understand the authentication mechanisms and plan their security testing accordingly.

## Exploitation Details

Since this alert is not a vulnerability, there are no specific exploitation details to provide.

## Step-by-Step Remediation

There is no technical remediation required for this alert, as it is an informational alert and not a vulnerability.

## References & Best Practices

- [OWASP ZAP Documentation](#)
- [OWASP Authentication Cheat Sheet](#)
- [OWASP Authentication Flaws](#)

This alert is a non-vulnerability alert and does not require any specific remediation. It is important to note that the information provided by this alert can be useful for security professionals to understand the authentication mechanisms used by the target application and plan their security testing accordingly.

---

## Retrieved from Cache

### Issue Explanation

The vulnerability in question is related to the disclosure of sensitive information in HTTP headers. This can be a significant security issue because it can lead to the leakage of sensitive data, such as session tokens, user-specific information, or personal details.

HTTP headers are part of the HTTP protocol and are used to pass additional information between the client and the server. The sensitive information can be included in various headers, such as `Set-Cookie`, `Authorization`, or `X-Forwarded-For`.

Attackers can use this information to perform various attacks, such as session hijacking, unauthorized access, or data theft. For example, if a session token is leaked in the `Set-Cookie` header, an attacker can use this token to impersonate the user and gain unauthorized access to their account.

### Impact Analysis

The impact of disclosing sensitive information in HTTP headers can be severe. It can lead to

the following security implications:

1. **Session Hijacking:** Attackers can use the leaked session tokens to hijack user sessions and gain unauthorized access to their accounts.
2. **Data Theft:** Sensitive data, such as personal information or user-specific details, can be exposed and potentially used for identity theft or other malicious purposes.
3. **Authorization Bypass:** If the `Authorization` header is leaked, attackers can bypass authentication mechanisms and gain unauthorized access to protected resources.
4. **Fingerprinting:** Attackers can use the leaked information to fingerprint users and tailor further attacks.
5. **Phishing and Social Engineering:** Attackers can use the leaked information to create more convincing phishing emails or social engineering attacks.

## Exploitation Details

To exploit this vulnerability, attackers can use various techniques such as:

1. **Automated Scanning:** Tools like Nmap, OWASP ZAP, or Burp Suite can be used to scan the target server and extract sensitive information from the HTTP headers.
2. **Manual Analysis:** Attackers can manually inspect the HTTP headers in the response from the server to identify sensitive information.
3. **Automated Header Collection:** Tools like `curl` or `wget` can be used to fetch the headers from the server and extract sensitive information.

Here's an example of how to use `curl` to collect headers and extract sensitive information:

```
curl -I http://example.com
```

This command will fetch the headers from the server and display them. The sensitive information can be found in various headers, such as `Set-Cookie`, `Authorization`, or `X-Forwarded-For`.

## Step-by-Step Remediation

To remediate this vulnerability, you can take the following steps:

### 1. Server-Level Remediation:

- **Apache:** Use the `Header edit` directive to remove sensitive headers or modify them to remove sensitive information. For example:

```
Header edit Set-Cookie ^(.*)$ $1; HttpOnly; Secure
```

- **Nginx:** Use the `more_set_headers` directive to remove sensitive headers or modify them to remove sensitive information. For example:

```
more_set_headers 'Set-Cookie: $http_cookie; HttpOnly; Secure';
```

- **IIS:** Use the `Set-Cookie` directive to set the `HttpOnly` and `Secure` flags to prevent client-side access to the cookie. For example:

```
Set-Cookie: sessionid=1234567890; HttpOnly; Secure
```

## 2. Application-Level Remediation:

- **Express.js:** Use the `res.cookie()` method with the `httpOnly` and `secure` options to set the cookie securely. For example:

```
res.cookie('sessionid', '1234567890', { httpOnly: true, secure:
```

- **PHP:** Use the `setcookie()` function with the `httponly` and `secure` parameters to set the cookie securely. For example:

```
setcookie('sessionid', '1234567890', time() + (86400 * 30), '/')
```

## References & Best Practices

- [OWASP ZAP Documentation](#)
  - [Nmap Documentation](#)
  - [Burp Suite Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

# Session Management Response Identified

## Issue Explanation

The alert "Session Management Response Identified" is an informational alert that indicates the presence of session management tokens in the HTTP response headers. This alert is not a vulnerability but rather a feature of the web application that can be used for session management.

Session management tokens are used to identify and manage user sessions on a web application. They are typically sent in the HTTP response headers and are used by the client to send back with subsequent requests to maintain the session state.

The "Other Info" field in the alert contains a set of header tokens that can be used for session management. If the web application is configured to use "Auto-Detect" for session management, it will automatically use these tokens for session management.

## Impact Analysis

The impact of this alert is not a security issue but rather a feature of the web application. It does not pose any direct security risk and does not need to be fixed. However, it is important to understand that the presence of session management tokens can be used by attackers to hijack user sessions if they are not properly secured.

## Exploitation Details

Since this is an informational alert and not a vulnerability, there are no specific exploitation details to provide.

## Step-by-Step Remediation

Since this is an informational alert, there is no technical remediation required. The presence of session management tokens is a feature of the web application and does not need to be fixed.

## References & Best Practices

- [OWASP ZAP Documentation](#)
  - [OWASP Session Management Cheat Sheet](#)
  - [OWASP Session Management Guide](#)
- 

## User Agent Fuzzer

### Issue Explanation

The vulnerability in question is related to the disclosure of server version information in HTTP headers. This can be a significant security issue because it can reveal the exact version of the web server software running on the target system. This information can be used by attackers to identify known vulnerabilities associated with that version, allowing them to craft targeted attacks.

HTTP headers are part of the HTTP protocol and are used to pass additional information between the client and the server. The server version information is typically included in the `Server` header, which is sent by the server to the client with each request. This header can contain details such as the server software name, version, and other metadata.

Attackers can use this information to fingerprint the server and identify potential vulnerabilities. For example, if the server version is known to be vulnerable to a specific exploit, the attacker can use this information to craft a malicious payload that exploits the vulnerability.

### Impact Analysis

The impact of disclosing server version information can be significant. It can lead to the following security implications:

1. **Fingerprinting:** Attackers can use the server version information to fingerprint the server and identify its exact version, which can be used to tailor further attacks.
2. **Version Enumeration:** Attackers can enumerate the exact version of the server software, which can be used to identify known vulnerabilities associated with that version.
3. **Combining with Other Reconnaissance Techniques:** The server version information

can be combined with other reconnaissance techniques, such as directory traversal, to gather more information about the server and its configuration.

4. **Targeted Attacks:** Attackers can use the server version information to craft targeted attacks that exploit known vulnerabilities specific to the identified version of the server software.
5. **Phishing and Social Engineering:** Attackers can use the server version information to create more convincing phishing emails or social engineering attacks, as they can tailor the attack to appear more legitimate.

## Exploitation Details

To exploit this vulnerability, attackers can use various techniques such as:

1. **Automated Scanning:** Tools like Nmap, OWASP ZAP, or Burp Suite can be used to scan the target server and extract the server version information from the `Server` header.
2. **Manual Analysis:** Attackers can manually inspect the HTTP headers in the response from the server to identify the server version.
3. **Automated Header Collection:** Tools like `curl` or `wget` can be used to fetch the headers from the server and extract the server version information.

Here's an example of how to use `curl` to collect headers and extract the server version:

```
curl -I http://example.com
```

This command will fetch the headers from the server and display them. The server version information can be found in the `Server` header.

## Step-by-Step Remediation

To remediate this vulnerability, you can take the following steps:

### 1. Server-Level Remediation:

- **Apache:** Set the `ServerTokens` directive to a minimal value to reduce the amount of information disclosed in the `Server` header. For example:

```
ServerTokens Prod
```

- **Nginx:** Set the `server_tokens` directive to `off` to disable the disclosure of server version information. For example:

```
server_tokens off;
```

- **IIS:** Set the `ServerHeader` directive to `off` to disable the disclosure of server version information. For example:

```
<system.webServer>
  <serverRuntime serverHeader="off" />
</system.webServer>
```

## 2. Application-Level Remediation:

- **Express.js:** Use the `trust proxy` middleware to set the `X-Forwarded-For` header, which can be used to hide the server version information. For example:

```
app.use(require('express').trustProxy());
```

- **PHP:** Use the `apache2handler` module to set the `ServerSignature` directive to `Off` to disable the disclosure of server version information. For example:

```
php_value serverSignature Off
```

## References & Best Practices

- [OWASP ZAP Documentation](#)
  - [Nmap Documentation](#)
  - [Burp Suite Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

# Cookie Without Secure Flag

## Issue Explanation

The vulnerability in question is related to the absence of the `Secure` flag in a cookie. This can be a significant security issue because it allows the cookie to be transmitted over unencrypted channels, such as HTTP, which can be intercepted by attackers.

Cookies are small pieces of data sent from a server to a user's browser and stored on the client-side. The `Secure` flag is a directive that tells the browser to only send the cookie over HTTPS connections, ensuring that the cookie is transmitted securely.

When a cookie is set without the `Secure` flag, it can be accessed via unencrypted HTTP connections, which can be intercepted by attackers. This can lead to the following security implications:

1. **Man-in-the-Middle (MitM) Attacks:** Attackers can intercept the unencrypted cookie and potentially gain unauthorized access to the user's session or sensitive information.
2. **Session Hijacking:** Attackers can hijack the user's session by stealing the cookie and using it to impersonate the user.
3. **Phishing and Social Engineering:** Attackers can use the cookie to create more convincing phishing emails or social engineering attacks, as they can use the stolen cookie to impersonate the user.

## Impact Analysis



The impact of not setting the `Secure` flag on cookies can be severe. It can lead to the following security implications:

1. **Sensitive Information Exposure:** Sensitive information stored in the cookie can be exposed to attackers, leading to data breaches.
2. **Session Hijacking:** Attackers can hijack user sessions, allowing them to access sensitive information or perform unauthorized actions.
3. **Phishing and Social Engineering:** Attackers can use the cookie to impersonate users, leading to further attacks.
4. **Lack of Confidentiality:** The lack of encryption can lead to a lack of confidentiality, as the cookie can be intercepted and read by anyone on the network.

## Exploitation Details

To exploit this vulnerability, attackers can use various techniques such as:

1. **Intercepting Unencrypted Traffic:** Attackers can intercept unencrypted HTTP traffic to capture the cookie.
2. **Session Hijacking:** Attackers can use the intercepted cookie to hijack the user's session.
3. **Phishing and Social Engineering:** Attackers can use the cookie to impersonate the user in phishing emails or social engineering attacks.

## Step-by-Step Remediation

To remediate this vulnerability, you can take the following steps:

### 1. Server-Level Remediation:

- **Apache:** Set the `Secure` flag for cookies using the `Set-Cookie` directive. For example:

```
Set-Cookie: mycookie=value; Secure
```

- **Nginx:** Set the `secure` parameter for cookies using the `add_header` directive. For example:

```
add_header Set-Cookie mycookie=value; Secure;
```

- **IIS:** Set the `Secure` flag for cookies using the `Set-Cookie` directive. For example:

```
Set-Cookie: mycookie=value; Secure
```

### 2. Application-Level Remediation:

- **Express.js:** Set the `secure` option for cookies using the `cookie` middleware. For example:

```
app.use(cookieParser({ secure: true }));
```

- **PHP:** Set the `secure` parameter for cookies using the `setcookie` function. For example:

```
setcookie('mycookie', 'value', time() + (86400 * 30), '/', 'example.com');
```

## References & Best Practices

- [OWASP ZAP Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

## Cookie without SameSite Attribute

### Issue Explanation

The vulnerability in question is related to the absence of the `SameSite` attribute in a cookie. The `SameSite` attribute is a browser-side policy that helps mitigate the risk of cross-site request forgery (CSRF) and cross-site scripting (XSS) attacks by controlling the contexts in which a cookie is sent.

Cookies without the `SameSite` attribute can be sent as a result of a 'cross-site' request, which can lead to security issues. For example, if an attacker tricks a user into visiting a malicious website, the attacker can potentially access the cookie and perform actions on behalf of the user without their knowledge.

### Impact Analysis

The impact of not setting the `SameSite` attribute can be significant:

1. **CSRF Attacks:** Attackers can perform CSRF attacks by tricking users into visiting a malicious website that sends a request to the server with the cookie, allowing the attacker to perform actions on behalf of the user.
2. **XSS Attacks:** Attackers can perform XSS attacks by injecting malicious scripts into the page, which can access the cookie and perform actions on behalf of the user.
3. **Session Hijacking:** Attackers can hijack user sessions by stealing the cookie and using it to impersonate the user.
4. **Data Breach:** Attackers can access sensitive data stored in the cookie, leading to a data breach.

### Exploitation Details

Attackers can exploit this vulnerability by:

1. **Phishing:** Sending phishing emails or links to users that trick them into visiting a

malicious website.

2. **Cross-Site Scripting (XSS):** Injecting malicious scripts into the page that can access the cookie.
3. **Cross-Site Request Forgery (CSRF):** Tricking users into performing actions on behalf of the attacker by sending requests with the cookie.

## Step-by-Step Remediation

To remediate this vulnerability, you can set the SameSite attribute to 'lax' or 'strict' for all cookies. Here's how you can do it for different platforms:

### 1. Server-Level Remediation:

- **Apache:** Set the SameSite directive in the Set-Cookie header. For example:  

```
Set-Cookie: mycookie=value; SameSite=lax
```
- **Nginx:** Set the same\_site directive in the add\_header directive. For example:  

```
add_header Set-Cookie mycookie=value; SameSite=lax;
```
- **IIS:** Set the SameSite attribute in the Set-Cookie header. For example:  

```
Set-Cookie: mycookie=value; SameSite=lax
```

### 2. Application-Level Remediation:

- **Express.js:** Set the SameSite attribute in the res.cookie() method. For example:  

```
res.cookie('mycookie', 'value', { sameSite: 'lax' });
```
- **PHP:** Set the SameSite attribute in the setcookie() function. For example:  

```
setcookie('mycookie', 'value', ['SameSite' => 'lax']);
```

## References & Best Practices

- [OWASP Documentation on SameSite](#)
  - [W3C Documentation on SameSite](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
- 

## Strict-Transport-Security Header Not Set

### Issue Explanation

The vulnerability in question is related to the absence of the HTTP Strict Transport Security

(HSTS) header. HSTS is a security feature that helps to protect websites against protocol downgrade attacks and cookie hijacking. It forces the client to communicate with the server over HTTPS only, even if the initial connection was made over HTTP.

The HSTS header is sent by the server to the client and instructs the client to only communicate with the server over HTTPS. This ensures that the connection is encrypted and prevents attackers from intercepting and tampering with the communication.

## Impact Analysis

The absence of the HSTS header can have the following security implications:

1. **Protocol Downgrade Attacks:** Attackers can intercept the initial HTTP connection and redirect the client to a malicious site, potentially leading to a man-in-the-middle attack.
2. **Cookie Hijacking:** Attackers can intercept and steal cookies that are not marked as secure, leading to session hijacking.
3. **Mixed Content Issues:** The absence of HSTS can lead to mixed content issues, where the client loads resources over HTTP instead of HTTPS, potentially exposing sensitive information.
4. **Bypassing Security Mechanisms:** Attackers can bypass security mechanisms that rely on the presence of the HSTS header, such as browser security warnings for non-HTTPS sites.

## Exploitation Details

Attackers can exploit the absence of the HSTS header by:

1. **Intercepting Initial Connections:** Attackers can intercept the initial HTTP connection and redirect the client to a malicious site, potentially leading to a man-in-the-middle attack.
2. **Stealing Cookies:** Attackers can steal cookies that are not marked as secure, leading to session hijacking.
3. **Bypassing Security Warnings:** Attackers can bypass browser security warnings for non-HTTPS sites, as the absence of the HSTS header does not trigger these warnings.

## Step-by-Step Remediation

To remediate this vulnerability, you can take the following steps:

### 1. Server-Level Remediation:

- **Apache:** Set the `Header always set Strict-Transport-Security` directive to enforce HSTS. For example:

```
Header always set Strict-Transport-Security "max-age=63072000;
```

- **Nginx:** Set the `add_header` directive to enforce HSTS. For example:

```
add_header Strict-Transport-Security "max-age=63072000; include
```

- **IIS:** Set the HSTS module to enforce HSTS. For example:

```
<system.webServer>
  <security>
    <hsts enabled="true" />
  </security>
</system.webServer>
```

## 2. Application-Level Remediation:

- **Express.js:** Use the `trust proxy` middleware to enforce HSTS. For example:

```
app.use(require('express').trustProxy());
```

- **PHP:** Use the `apache2handler` module to enforce HSTS. For example:

```
php_value hsts on
```

## References & Best Practices

- [OWASP HSTS Documentation](#)
  - [Apache HTTP Server Documentation](#)
  - [Nginx Documentation](#)
  - [IIS Documentation](#)
  - [Express.js Documentation](#)
  - [PHP Documentation](#)
-