**Année universitaire 2024-2025**

 **Ethical hacking and defense Integrated Project**

**Cyber Security Major**

**Coordinator : Pr. Anass Sebbar**

**CO-Advisor : Ahmed Didouh**

**Othmane Cherqi**

**Student : Ayman Mourtada**

**Web Application Security :**

**Analysis & Defense Against**

**Header Manipulation**

**INTRODUCTION**

In today's interconnected digital landscape, web application security stands as a cornerstone of cybersecurity defence. As organizations increasingly rely on web-based services, understanding and protecting against sophisticated attack vectors becomes paramount for security professionals and developers alike. This project focuses on one particularly nuanced yet potent attack vector: HTTP Header Manipulation, with a special emphasis on Host Header Attacks.

Header manipulation in web applications has become a critical security concern in modern web development. While HTTP headers serve essential functions in web communication, they can also become vectors for sophisticated attacks when improperly handled or validated.

One particularly concerning attack vector is HTTP Host Header manipulation, where attackers modify the Host header in HTTP requests to exploit web applications. This technique can lead to severe security breaches by making the server believe requests originate from trusted domains when they actually don't.

These attacks are particularly dangerous in shared hosting environments, where multiple websites reside on the same server. If a web application relies on the **Host** header for security decisions (generating URLs or validating requests), an attacker can exploit this to inject malicious content, redirect users to fraudulent sites, or perform unauthorized actions.

Web application security is a cornerstone of modern software development, as applications grow in complexity and become prime targets for malicious actors. Header manipulation is a common attack vector, enabling techniques like clickjacking, XSS, and CSRF. To understand and mitigate these risks, we will explore the concept of header manipulation through a hands-on project.

**Research Scope**

**What Is HTTP Header?**

**HTTP Host Header**  refer to a type of web application attack where an attacker manipulates the Host header field in an HTTP request to inject malicious content into a web page or redirect the user to a different website

By modifying this header, attackers can deceive servers into believing requests originate from trusted domains, potentially leading to severe security breaches including unauthorized access, data theft, and service disruption. This attack vector is particularly potent in shared hosting environments where multiple websites coexist on a single server, as it can be exploited to bypass security measures, inject malicious content, redirect users to fraudulent websites, and facilitate advanced attacks such as cross-site scripting (XSS), cross-site request forgery (CSRF), and sophisticated phishing campaigns.

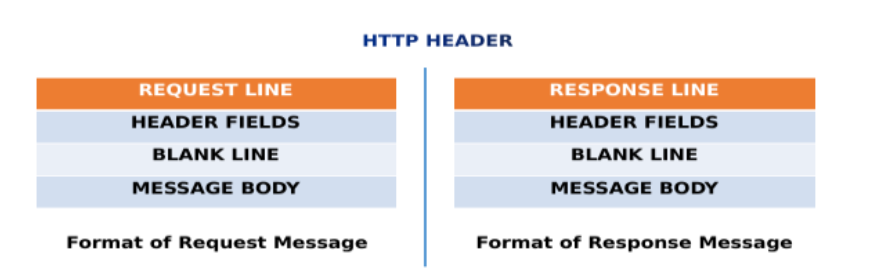
HTTP headers let the client and the server pass additional information with an HTTP request or response.

An HTTP header is defined by a case insensitive name followed by a colon**(:)** followed by its value example, Host: **www.test.com**

Headers can be grouped into four types according to their context, which is defined as follows:

* General Headers
* `Request Headers A line with text on it

  Description automatically generated
* Response Headers
* Entity Headers



**What Is X-Forwarded-Host?**

X-Forwarded-Host header is used to identify the original request initiated by the client. Since hostnames and ports differ in reverse proxies, this header is used to identify the original request. This header is also used for debugging purposes.

Syntax: X-Forwarded-Host: www.site.com

**Example of vulnerable code on different programming languages:**

* In PHP

**A computer screen shot of white text

Description automatically generated**

In the above PHP code, the server redirects the user to the home page if the Host header is equal to “example.com”. However, an attacker can manipulate the Host header to bypass this check and redirect the user to a malicious website

* In JAVA

A screen shot of a computer code

Description automatically generated

In the above Java code, the server redirects the user to the home page if the Host header is equal to “example.com”. However, an attacker can manipulate the Host header to bypass this check and redirect the user to a malicious website.

* In Python

A computer screen shot of a program

Description automatically generated

In the above Python code, the server redirects the user to the home page if the Host header is equal to “example.com”. However, an attacker can manipulate the Host header to bypass this check and redirect the user to a malicious website.

**Examples of exploitation HTTP Host Header Attacks**

**Phishing**: An attacker can manipulate the Host header to make the server believe that a request is coming from a trusted domain, such as a bank’s website. The attacker can then redirect the user to a fake login page that looks like the bank’s website, steal their login credentials, and use them to carry out fraudulent activities.

**Cross-site scripting (XSS)**:

An attacker can inject malicious JavaScript code into a web page by manipulating the Host header. When the user visits the page, the JavaScript code executes in their browser, allowing the attacker to steal their sensitive information, such as cookies or login credentials.

**Cross-site request forgery (CSRF)**:

An attacker can manipulate the Host header to trick the user’s browser into sending a request to a vulnerable web application that is authenticated with the user’s credentials. The attacker can then carry out various malicious activities, such as changing the user’s password, making unauthorized transactions, or deleting their data.

**Server-side request forgery (SSRF)**:

An attacker can manipulate the Host header to make the server send requests to internal or external resources that are not intended to be accessed by the web application. This can lead to information disclosure, remote code execution, or denial of service attacks.

**Clickjacking:**  
In this technique, the attacker overlays a transparent layer over a website, which can be used to trick the user into clicking on a hidden link or button. The attacker can then use this technique to steal user credentials or perform actions on behalf of the victim.

**List of payloads HTTP Host Header Attacks**

* **Malicious domain names:** Attackers can craft requests with malicious domain names to trick the server into sending sensitive data to a domain under the attacker’s control.
* **IP addresses:** Attackers can use IP addresses in the Host header to bypass access controls that rely on domain name matching.

**Localhost:** Attackers can use “localhost” or “127.0.0.1” in the Host header to bypass access controls that restrict access to local services.

* **URL path traversal:**Attackers can use the Host header to perform URL path traversal attacks, allowing them to access files or directories outside of the intended location.
* **Port scanning:** Attackers can use the Host header to scan for open ports on the server.
* **Cross-site scripting (XSS):** Attackers can use the Host header to inject malicious scripts into web pages.
* **SQL injection:** Attackers can use the Host header to inject SQL commands into database queries.
* **Buffer overflow:** Attackers can use the Host header to trigger buffer overflow vulnerabilities in web applications.
* **File inclusion:** Attackers can use the Host header to include arbitrary files on the server.
* **Server-side request forgery (SSRF):** Attackers can use the Host header to trigger SSRF vulnerabilities in web applications, allowing them to make requests to internal or external resources on behalf of the server

**Project Framework**

**Overview**

A Web application security is a cornerstone of modern software development, as applications grow in complexity and become prime targets for malicious actors. One of the most overlooked yet critical attack vectors is **header manipulation**, where attackers exploit insecure or misconfigured HTTP headers to perform attacks such as clickjacking, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and more. This project delves into the intricacies of header manipulation, demonstrating how attackers exploit vulnerabilities and how developers can mitigate these risks.

The project is structured into three phases:

* **Vulnerable Web Application** : A Flask-based application intentionally left vulnerable to header manipulation attacks.
* **Attack Demonstration**: Exploitation of vulnerabilities using techniques like clickjacking, XSS, and CSRF.
* **Secured Web Application :** Implementation of robust security measures to mitigate the demonstrated attacks

**Methodology**

**Development of the Low-Security Web Application**

The **app.py** file represents a simple Flask-based blog application with intentional vulnerabilities. Key features include:

* **Database Models**: SQLite database storing posts, categories, and tags.
* **Routes**: Endpoints for viewing posts, categories, tags, and performing searches.
* **Templates**: Jinja2 templates for rendering HTML pages

A screenshot of a computer program

Description automatically generated

* **Key Configurations:**
* **Database URI**: The application uses SQLite (sqlite:///blog.db) for the database.
* **Track Modifications**: Disabled to suppress warnings (SQLALCHEMY\_TRACK\_MODIFICATIONS = False)

A screenshot of a computer screen

Description automatically generated

* **Routes**

The application defines several routes to handle different views.

**Home Route (/)**

Displays a paginated list of blog posts, ordered by the most recent .

**Post Route (/post/<slug>)**

Displays a single blog post and related posts from the same category.

**Category Route (/category/<slug>)**

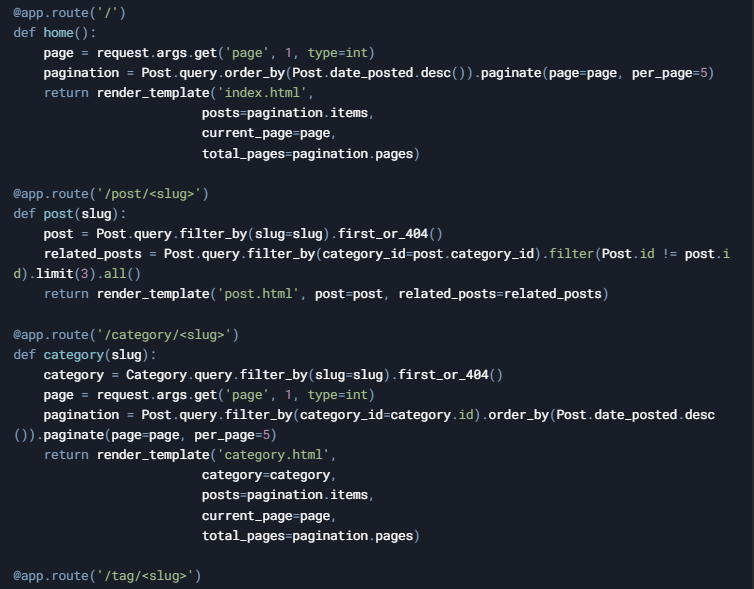
Displays all posts belonging to a specific category, paginated.

**Tag Route (/tag/<slug>)**

Displays all posts associated with a specific tag, paginated.

**Search Route (/search)**

Allows users to search for posts by title or content.



* **Running the Application**

To run the application, execute the script:

python app.py

The application will start in debug mode, and the database will be initialized with sample data. Access the application at <http://127.0.0.1:5000>.

* **User Interface Development**

**UI Design**: The user interface (UI) was designed using basic HTML and CSS, with simple forms to interact with the application While the UI itself was not complex, it was sufficient to simulate user interactions and showcase how security vulnerabilities could be exploited.

**Interface 1: Homepage**

A screenshot of a computer

Description automatically generated

**Interface 2: Post or Category View**

A screenshot of a computer

Description automatically generated

* **Security Issues in app.py**
* **Lack of Security Headers**: Missing headers like X-Frame-Options, Content-Security-Policy, and X-Content-Type-Options.
* **No HTTPS Enforcement**: Absence of HTTPS makes the application susceptible to man-in-the-middle (MITM) attacks.\
* **Debug Mode Enabled**: Debug mode exposes sensitive information during errors.
* **Absence of CSP Configurations**: The app did not have a Content-Security-Policy **(CSP)**, leaving it open to XSS attacks and unauthorized content injection.

**Attack Implementation & Testing**

**Clickjacking Attack Development**

The clickjacking.html file demonstrates a **clickjacking attack** by overlaying a transparent iframe on top of the vulnerable web application. The user is tricked into clicking a button that appears to be part of the legitimate website but is actually part of the malicious page.



**How the Attack Works**

1. **Transparent Iframe**:
   * The iframe is made transparent using CSS (opacity : 0.9), making it difficult for the user to notice.
2. **Overlay Button**:
   * A button labeled "Claim Your Prize" is placed on top of the iframe. When clicked, it triggers a JavaScript function that simulates a successful action.
3. **User Deception**:
   * The user believes they are interacting with the legitimate website, but their actions are being captured by the malicious page

A screenshot of a computer

Description automatically generated

**Man-in-the-Middle (MITM) Attack**

**What is a Man-in-the-Middle Attack?**

A Man-in-the-Middle (MITM) attack occurs when an attacker intercepts and potentially alters the communication between two parties without their knowledge. This can lead to data theft, session hijacking, or unauthorized actions.

**Demonstration: MITM Attack**

To demonstrate a MITM attack, we can use a tool like Wireshark or Burp Suite to intercept and analyze the traffic between the client and the server. However, for this demonstration, we will focus on how the lack of HTTPS enforcement in the vulnerable application (app.py) makes it susceptible to MITM attacks**.**

**A screenshot of a computer

Description automatically generated**

**Mitigation**

Enforce HTTPS using **Flask-Talisman**:

**A computer screen shot of white text

Description automatically generated**

**X-Forwarded-Host Manipulation**

The X-Forwarded-Host header is used by proxies to forward the original host. If not properly validated, it can be manipulate

**A screen shot of a computer

Description automatically generated**

**Exploitation:**

* An attacker can inject a malicious X-Forwarded-Host header:

curl -H "X-Forwarded-Host: attacker.com" m

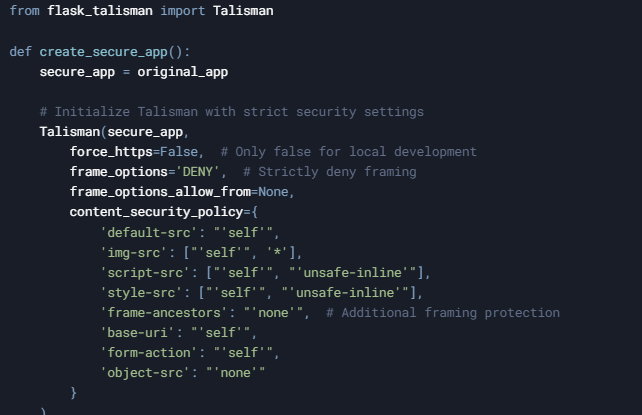
**Securing the Web Application**

The **appsec.py** file is a hardened version of **app.py**, incorporating the following security enhancements:

* **Flask-Talisman Integration**:

**Key Features of Flask-Talisman:**

* + **Content Security Policy (CSP):** Prevents unauthorized execution of scripts and other resources.
  + **X-Frame-Options**: Protects against clickjacking by controlling how the application can be embedded in iframes.
  + **Strict Transport Security (HSTS):** Ensures that the application is only accessed over HTTPS.
  + **X-Content-Type-Options:** Prevents MIME sniffing by forcing the browser to respect the declared content type.



**frame\_options='DENY'**: This setting prevents the application from being embedded in an iframe, effectively mitigating clickjacking attacks.

* **Custom Security Headers**:
  + Additional headers such as X-XSS-Protection, Referrer-Policy, and Permissions-Policy are added to further enhance security.

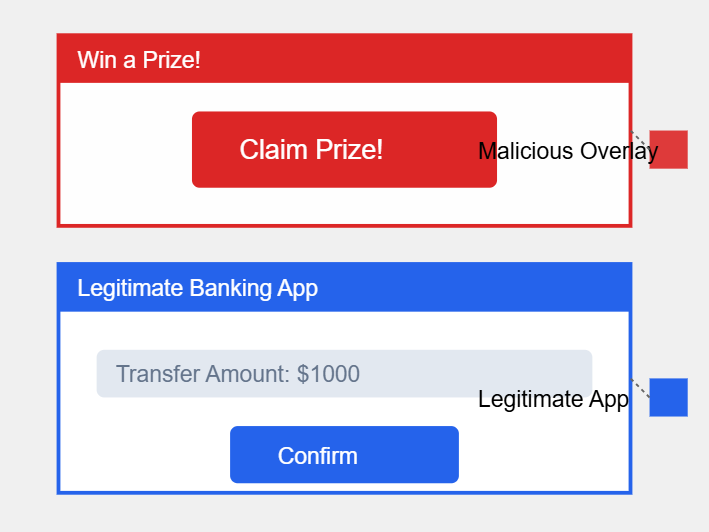
A screen shot of a computer program

Description automatically generated

* **Clickjacking Protection**:
  + The X-Frame-Options header is set to DENY, preventing the application from being embedded in an iframe.
  + The Content-Security-Policy header includes frame-ancestors 'none', which also prevents framing.

Mitigation with **appsec.py**:

After securing the application with appsec.py, the clickjacking attack is no longer possible. The X-Frame-Options: DENY and Content-Security-Policy: frame-ancestors 'none' headers prevent the application from being embedded in an iframe, effectively blocking the attack.



* **Additional Security Measures :**

**HTTPS Enforcement**

While appsec.py sets force\_https=False for local development, it is crucial to enforce HTTPS in production. This prevents man-in-the-middle (MITM) attacks and ensures that all data transmitted between the client and server is encrypted.

**Debug Mode Disabled**

In production, the application should not run in debug mode. Debug mode can expose sensitive information, such as stack traces, in case of errors. The appsec.py file ensures that the application runs with debug=False.

**Secure Session Management**

If the application includes user authentication, it is essential to configure secure session management. This includes:

* Setting SESSION\_COOKIE\_SECURE to ensure cookies are only sent over HTTPS.
* Setting SESSION\_COOKIE\_HTTPONLY to prevent JavaScript from accessing cookies.

**Mitigation with appsec.py:**

**clickjacking attack**

After securing the application with appsec.py, the **clickjacking attack** is no longer possible. The X-Frame-Options: DENY and Content-Security-Policy: frame-ancestors 'none' headers prevent the application from being embedded in an iframe, effectively blocking the attack.

**MITM Attack**

The appsec.py file includes several measures to prevent MITM attacks:

1. **HTTPS Enforcement**:
   * The force\_https option in Flask-Talisman ensures that all traffic is encrypted using HTTPS.
   * In production, this setting should be enabled to prevent attackers from intercepting plaintext traffic.
2. **Secure Cookies**:
   * The SESSION\_COOKIE\_SECURE setting ensures that cookies are only sent over HTTPS, preventing attackers from capturing session cookies.
3. **HTTP Strict Transport Security (HSTS)**:
   * The Strict-Transport-Security header instructs the browser to only access the application over HTTPS, even if the user types http:// in the address bar.

A screen shot of a computer

Description automatically generated

**Comparison of Vulnerable vs. Secured Web Application**

A screenshot of a computer

Description automatically generated

**Security Header Table**

A screenshot of a black screen

Description automatically generated

**Color Legend**

* **Red:** High Risk - Critical for protecting against high-impact vulnerabilities like XSS, clickjacking, and MITM attacks.
* **Yellow:** Medium Risk - Important for mitigating specific attacks but not as critical as the red-level protections.
* **Green:** Low Risk - A solid defense that provides good protection but may not be as urgently necessary as others.

**How to be protected from HTTP Host Header Attacks**

* **Validate input:** Ensure that all input received by the server is validated and sanitized before being used.
* **Use strict domain name matching:** Ensure that access controls rely on strict domain name matching, and not just the Host header value.
* **Whitelist allowed domain names:** Maintain a whitelist of allowed domain names, and reject requests with domain names not on the list.
* **Use SSL/TLS:** Ensure that all web traffic is encrypted using SSL/TLS, to prevent attackers from intercepting or modifying requests.
* **Set a default virtual host:** Set a default virtual host that will handle all requests with unrecognized domain names, to prevent them from being routed to unintended resources.
* **Avoid using IP addresses in the Host header:** Avoid using IP addresses in the Host header, as this can allow attackers to bypass access controls that rely on domain name matching.
* **Limit server responses:** Limit the amount of information included in server responses, to minimize the risk of sensitive data exposure.
* **Implement strict access controls:** Implement strict access controls to limit the resources that can be accessed based on user permissions.
* **Keep software up to date:** Keep all software and web applications up to date with the latest security patches and updates, to minimize the risk of known vulnerabilities being exploited.
* **Perform regular security testing:** Perform regular security testing, including vulnerability scanning and penetration testing, to identify and address any potential security weaknesses in the system.

**Mitigations for HTTP Host Header Attacks**

* **Use a web application firewall (WAF):** A WAF can be configured to block requests with invalid Host headers or to filter out requests that contain malicious payloads.
* **Perform input validation and sanitization:** Validate and sanitize input from all sources, including the Host header, to ensure that it contains only expected characters and is not used to execute malicious commands.
* **Use secure coding practices:** Ensure that web application code is written with security in mind, following best practices for secure coding such as using prepared statements, parameterized queries, and stored procedures.
* **Use HTTPS:** Use HTTPS to encrypt all traffic between clients and servers, which can help prevent man-in-the-middle attacks that could modify the Host header in transit.
* **Use server-side redirection:** Use server-side redirection to enforce that all requests are directed to a single, canonical hostname. This can help prevent attacks that rely on requests being processed with different Host headers.
* **Use subdomain isolation:** If possible, isolate different parts of a web application onto different subdomains. This can limit the impact of attacks that exploit the Host header by restricting access to sensitive resources to specific subdomains.
* **Limit exposure:** Limit the exposure of sensitive resources by implementing access control mechanisms and only exposing resources that are necessary for the operation of the application.

**Advanced Header Manipulation Defense Strategy**

**1. Simple Host Header Validation**

Quick implementation to validate Host headers:

A screen shot of a computer program

Description automatically generated

This middleware:

* Checks incoming Host headers
* Allows only specified domains
* Blocks requests with invalid hosts

**2. Header Injection Protection**

Basic protection against header injection:

A screen shot of a computer code

Description automatically generated

**Key Protection Tips:**

1. **Always Validate Headers**
   * Check lengths
   * Validate characters
   * Verify domains
2. **Set Security Headers**
   * X-Frame-Options
   * Content-Security-Policy
   * X-XSS-Protection
3. **Use HTTPS**
   * Force HTTPS in production
   * Implement HSTS
4. **Monitor and Log**
   * Track suspicious patterns
   * Log failed attempts
   * Set up alerts

**Conclusion**

This project highlighted the critical importance of HTTP header configuration in web application security. By starting with a vulnerable Flask-based application **(app.py)** and progressively securing it **(appsec.py)** , we demonstrated how simple misconfigurations can lead to severe vulnerabilities. The implementation of tools like Flask-Talisman and custom security headers significantly enhanced the application's security, protecting against common attacks like clickjacking, XSS, and CSRF.

As web applications continue to evolve, so too must our approach to securing them. By adopting a proactive and comprehensive security strategy, developers can build applications that are resilient to emerging threats and provide a safe experience for users

**References**

* **OWASP Top Ten Project:**[**https://owasp.org/www-project-top-ten/**](https://owasp.org/www-project-top-ten/)
* **OWASP Clickjacking Defense Cheat Sheet:**[**https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking\_Defense\_Cheat\_Sheet.html**](https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html)
* **OWASP CSRF Prevention Cheat Sheet:**[**https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\_Request\_Forgery\_Prevention\_Cheat\_Sheet.html**](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)
* **OWASP XSS Prevention Cheat Sheet:**[**https://cheatsheetseries.owasp.org/cheatsheets/Cross\_Site\_Scripting\_Prevention\_Cheat\_Sheet.html**](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html)
* **MDN Web Docs: HTTP Headers:**[**https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers)
* **MDN Web Docs: CSP:**[**https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP)
* **MDN Web Docs: HSTS:**[**https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security)
* **Burp Suite:**[**https://portswigger.net/burp**](https://portswigger.net/burp)
* **Postman:**[**https://www.postman.com/**](https://www.postman.com/)
* **Wireshark:**[**https://www.wireshark.org/**](https://www.wireshark.org/)
* **OWASP ZAP:**[**https://www.zaproxy.org/**](https://www.zaproxy.org/)
* **Flask Documentation:**[**https://flask.palletsprojects.com/**](https://flask.palletsprojects.com/)
* **Flask-Talisman Documentation:**[**https://flask-talisman.readthedocs.io/**](https://flask-talisman.readthedocs.io/)
* **Flask-Security Documentation:**[**https://flask-security.readthedocs.io/**](https://flask-security.readthedocs.io/)
* **Google Web Fundamentals: Security:**[**https://web.dev/security/**](https://web.dev/security/)
* **HTTP Host Header Attacks:**[**https://portswigger.net/web-security/host-header**](https://portswigger.net/web-security/host-header)
* **Mozilla Observatory:**[**https://observatory.mozilla.org/**](https://observatory.mozilla.org/)
* **SecurityHeaders.com:**[**https://securityheaders.com/**](https://securityheaders.com/)
* **CSP Evaluator:**[**https://csp-evaluator.withgoogle.com/**](https://csp-evaluator.withgoogle.com/)
* **HSTS Preload List:**[**https://hstspreload.org/**](https://hstspreload.org/)