RESEARCH REPORT

CROSS-ORIGIN RESOURCE SHARING (CORS)

&

HTTP STRICT TRANSPORT SECURITY (HSTS)

Guide: Mr Ali J & Mr Kuldeep L M

Jayashankar P _ Spyder 9

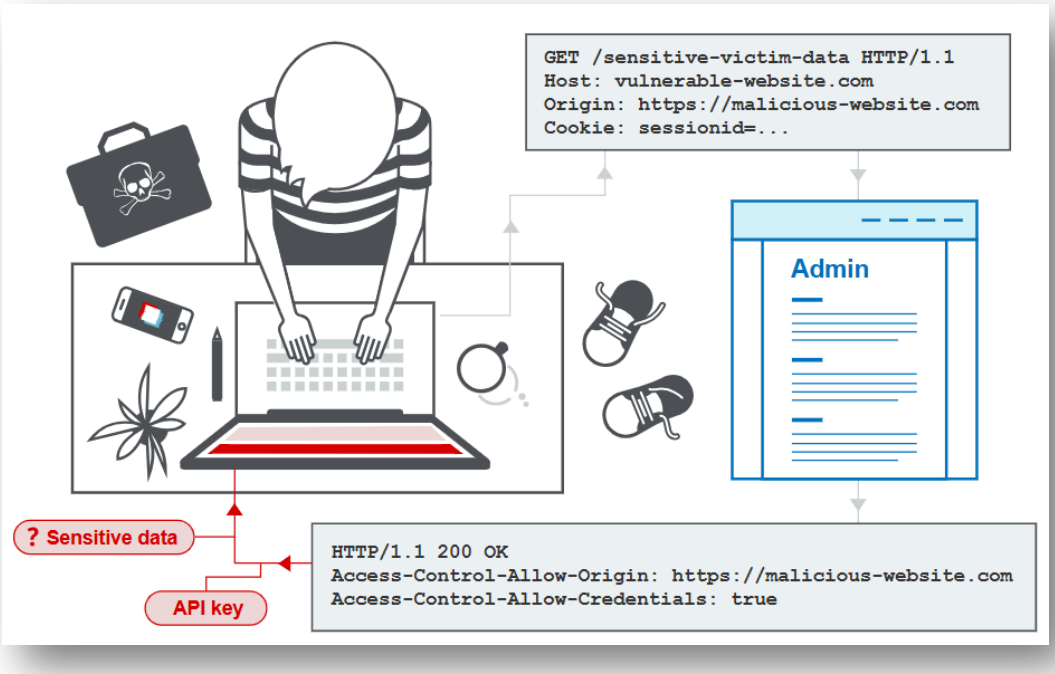
Red Team Intern @ Cyber Sapiens

6 . 1 . 2025

CROSS-ORIGIN RESOURCE SHARING (CORS)

INTRODUCTION TO CORS

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as cross-site request forgery (CSRF).



Let's understand this in simple way;

Imagine this: You're at a party, and you want to grab a drink from someone else's cooler.

- ✓ Without CORS: You can't just take a drink. It's like a security rule – you can only drink from your own cooler. This is the "same-origin policy" in web browsing.
- ✓ With CORS: The party host can give you permission to take a drink from their cooler. They might say, "Sure, you can have a soda, but no beer!" This is like CORS – it allows controlled access to resources from a different website.

In simple terms:

CORS is a system that lets websites share data with each other safely. It's like a set of rules that say who can access what information. This helps keep your data secure while still allowing useful things like:

- ✓ Using maps from Google Maps on your website: Your website needs to access data from Google's servers.
- ✓ Logging in with your Facebook account: Your website needs to share some information with Facebook's servers.
- ✓ Displaying tweets from Twitter: Your website needs to get data from Twitter's servers.

CORS makes these things possible by letting websites communicate with each other in a controlled way.

TYPES OF CORS REQUESTS

CORS requests can be classified primarily into two types: simple requests and preflight requests. Each type has specific characteristics and serves different purposes within the CORS framework.

Simple Requests

Simple requests are certain kinds of requests that are considered safe enough not to require explicit CORS preflight approval. These requests include:

Characteristics	Examples
Use of HTTP methods: GET, POST, or HEAD. Only certain headers are allowed, such as Accept, Content-Language, Content-Type with certain values, and a few others. No custom headers or ReadableStream objects in requests.	A GET request for a public image or a JSON file from a different domain. A POST request submitting a form to a server from a different domain, where the Content-Type is application/x-www-form-urlencoded, multipart/form-data, or text/plain. These requests are considered 'simple' because they don't significantly differ from typical HTTP requests and are unlikely to pose a security risk.

Example Request >>

```
GET /resource/example
Origin: http://example.com
```

Server Response >>

```
Access-Control-Allow-Origin: http://example.com
```

Preflight Requests

Preflight requests are used for more complex or potentially risky requests, providing an additional layer of security. These include:

Characteristics	Example
Involve HTTP methods other than GET, POST, or HEAD, such as PUT, DELETE, or CONNECT. Use of custom headers or non-standard content types. Requests that might modify data on the server.	A PUT request to update a record in a database hosted on a different domain, where the request includes custom headers like X-My-Custom-Header. Preflight requests are essential to CORS, ensuring that the server explicitly permits more complex or potentially risky cross-origin requests before they are made. This mechanism helps maintain web security while still allowing the necessary flexibility for modern web applications.

Example Request >>

```
OPTIONS /resource/example
Origin: http://example.com
Access-Control-Request-Method: PUT
```

Server Response >>

```
Access-Control-Allow-Origin: http://example.com
Access-Control-Allow-Methods: PUT
```

Credentialed Requests

- ✓ Requests with credentials (cookies, HTTP authentication, & client-side SSL certificates)
- ✓ Require the server to respond with not only Access-Control-Allow-Origin, but also Access-Control-Allow-Credentials: true.

Example Request >>

```
GET /resource/example
Origin: http://example.com
Cookie: sessionId=abc123
```

Server Response >>

```
Access-Control-Allow-Origin: http://example.com
Access-Control-Allow-Credentials: true
```

How CORS configuration will get vulnerable ?

CORS (Cross-Origin Resource Sharing) vulnerabilities arise from misconfigurations in how web servers handle requests from different domains. Here's how these vulnerabilities can occur:

Permissive Header / Allowing any origin:

- ✓ Wildcard Misuse: Setting Access-Control-Allow-Origin to * grants access to any domain, making the resource vulnerable to attacks from malicious websites.
- ✓ Reflected Origins: Dynamically setting Access-Control-Allow-Origin to the value of the Origin request header can be exploited by attackers to gain access.

Missing or Incorrect Headers:

- ✓ Missing Access-Control-Allow-Credentials: If an application needs to send cookies or HTTP authentication credentials with cross-origin requests, the Access-Control-Allow-Credentials header must be set to true. Omitting this header prevents credentials from being sent, potentially limiting functionality.
- ✓ Incorrect Access-Control-Allow-Methods or Access-Control-Allow-Headers: These headers specify the allowed HTTP methods (**e.g., GET, POST, PUT**) and custom headers for cross-origin requests. Incorrect configurations can restrict legitimate requests.

CORS Preflight Requests:

- ✓ Incorrect Handling of Preflight OPTIONS Requests: For certain request methods (like POST with custom headers), browsers send a preflight OPTIONS request to check if the cross-origin request is allowed. If the server doesn't handle these requests correctly, it can block legitimate requests.

Insecure Credential Sharing:

- ✓ Allowing Credentials with *: If Access-Control-Allow-Credentials is set to true while using * for Access-Control-Allow-Origin, attackers can potentially steal sensitive data like cookies or authentication tokens.

Exploiting CORS Vulnerability ?

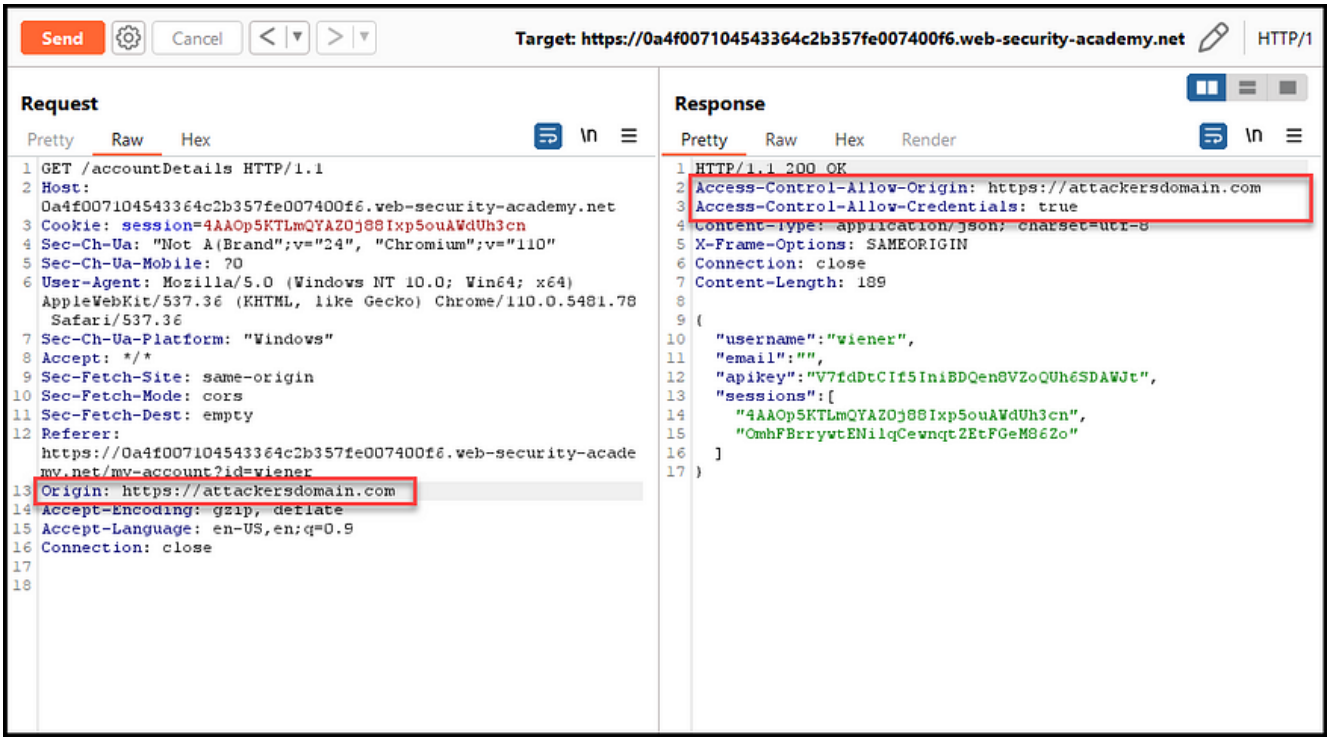
To identify CORS issues, we modify the Origin header in the requests with multiple values and see what response headers we get back from the application. There are four (4) known ways to do so:

1. Reflected Origins

Set the Origin header in the request to an arbitrary domain, such as https://attackersdomain.com, and check the Access-Control-Allow-Origin header in the response. If it reflects the exact domain you supplied in the request, it means the domain doesn't filter for any origins.

The risk of this misconfiguration is high risk if the domain allows for credentials to be passed in the requests. We can validate that by checking if the Access-Control-Allow-Credentials header is also included in the response and is set to true.

However, the risk is low if passing credentials is not allowed, as the browser will not process the responses from authenticated requests.



2. Modified Origins

Set the Origin header to a value that matches the targeted domain, but add a prefix or suffix to the domain to check if there is any validation on the beginnings or ends of the domain.

If no checks are in place, we can create a similar matching domain that bypasses the CORS policy on the targeted domain. For example, adding a prefix or suffix to the metrics.com domain would be something like attackmetrics.com or metrics.com.attack.com.

The risk of this misconfiguration is considered high risk if the domain allows for passing credentials with the Access-Control-Allow-Credentials header set to true, as the attacker can create a similar matching domain and retrieve sensitive information from the targeted domain.

However, the risk would be low if authenticated requests were not allowed.

3. Trusted subdomains with Insecure Protocol.

Set the Origin header to an existing subdomain and see if it accepts it. If it does, it means the domain trusts all its subdomains, which is not a good idea because if one of the subdomains has a Cross-Site Scripting (XSS) vulnerability, it will allow the attacker to inject a malicious JS payload and perform unauthorized actions.

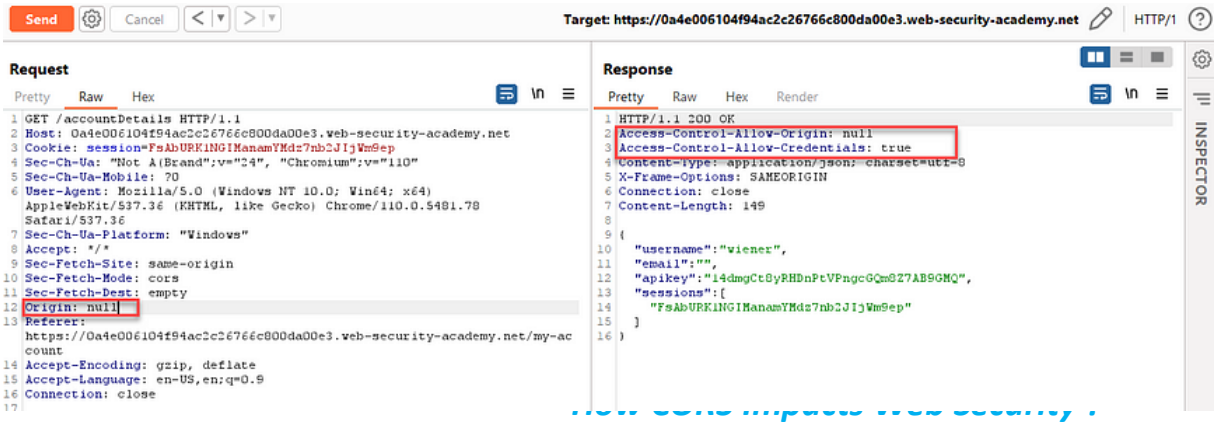
This misconfiguration is considered high risk if the domain accepts subdomains with an insecure protocol, such as HTTP, and the credential header is set to true. Otherwise, it will not be exploitable and would be only a poor CORS implementation.

4. Null Origin

Set the Origin header to the null value — Origin: null, and see if the application sets the Access-Control-Allow-Origin header to null. If it does, it means that null origins are whitelisted.

The risk level is considered high if the domain allows for authenticated requests with the Access-Control-Allow-Credentials header set to true.

However, if it does not, then the issue is considered low, and not exploitable.



CORS misconfigurations can have a significant impact on the security of web applications. Below are the main implications:

- Data Theft: Attackers can use CORS vulnerabilities to steal sensitive data from applications like API keys, SSH keys, Personal identifiable information (PII), or users' credentials.
- Cross-Site Scripting (XSS): Attackers can use CORS vulnerabilities to perform XSS attacks by injecting malicious scripts into web pages to steal session tokens or perform unauthorized actions on behalf of the user.

How to mitigate the CORS Vulnerability ?

CORS vulnerabilities arise primarily as misconfigurations. Prevention is therefore a configuration problem. The following sections describe some effective defenses against CORS attacks.

- Implement proper CORS headers: The server can add appropriate CORS headers to allow cross-origin requests from only trusted sites.
- If a web resource contains sensitive information, the origin should be properly specified in the Access-Control-Allow-Origin header.
- Restrict access to sensitive data: It is important to restrict access to sensitive data to only trusted domains. This can be done by implementing access control measures such as authentication and authorization.
- Avoid using the header Access-Control-Allow-Origin: null. Cross-origin resource calls from internal documents and sandboxed requests can specify the null origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.
- Avoid using wildcards in internal networks. Trusting network configuration alone to protect internal resources is not sufficient when internal browsers can access untrusted external domains.

EXPLOITING CORS VULNERABILITY

LAB 1: CORS Vulnerability with basic origin reflection

URL: https://portswigger.net/web-security/cors/lab-basic-origin-reflection-attack

GOAL: This website has an insecure CORS configuration in that it trusts all origins.

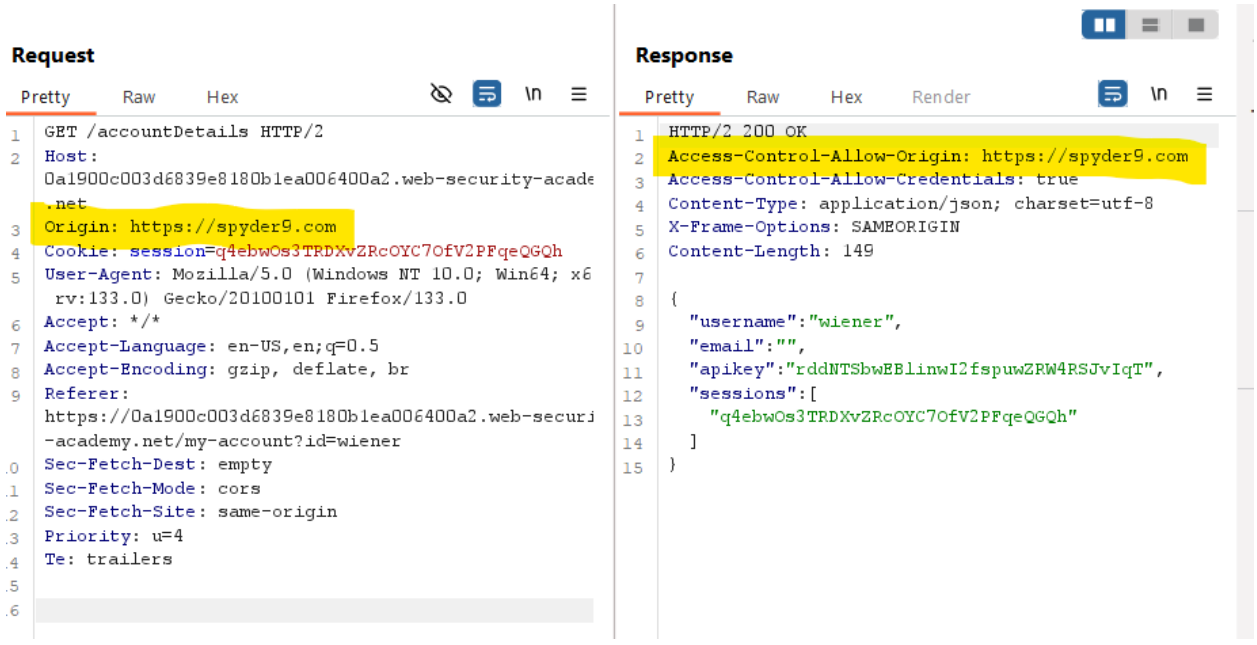
To solve the lab, craft some JavaScript that uses CORS to retrieve the administrator's API key and upload the code to your exploit server. The lab is solved when you successfully submit the administrator's API key.

You can log in to your own account using the following credentials: wiener:peter

SOLVING STEPS:

Check intercept is off, then use the browser to log in and access your account page.

Review the history and observe that your key is retrieved via an AJAX request to /accountDetails, and the response contains the Access-Control-Allow-Credentials header suggesting that it may support CORS.



Send the request to Burp Repeater, and resubmit it with the added header:

Origin: https://spyder9.com

Observe that the origin is reflected in the Access-Control-Allow-Origin header.

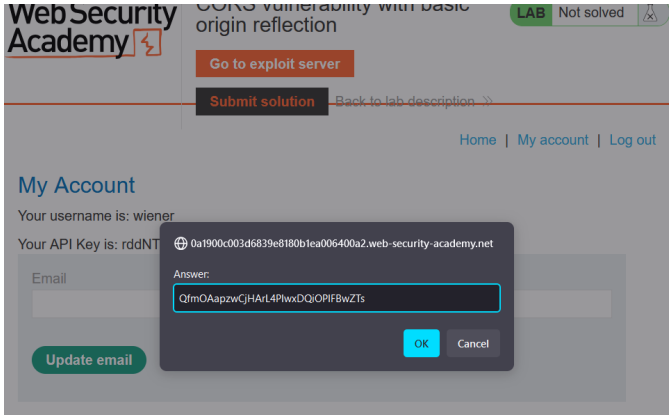
In the browser, go to the exploit server and enter the following HTML, replacing YOUR-LAB-ID with your unique lab URL:

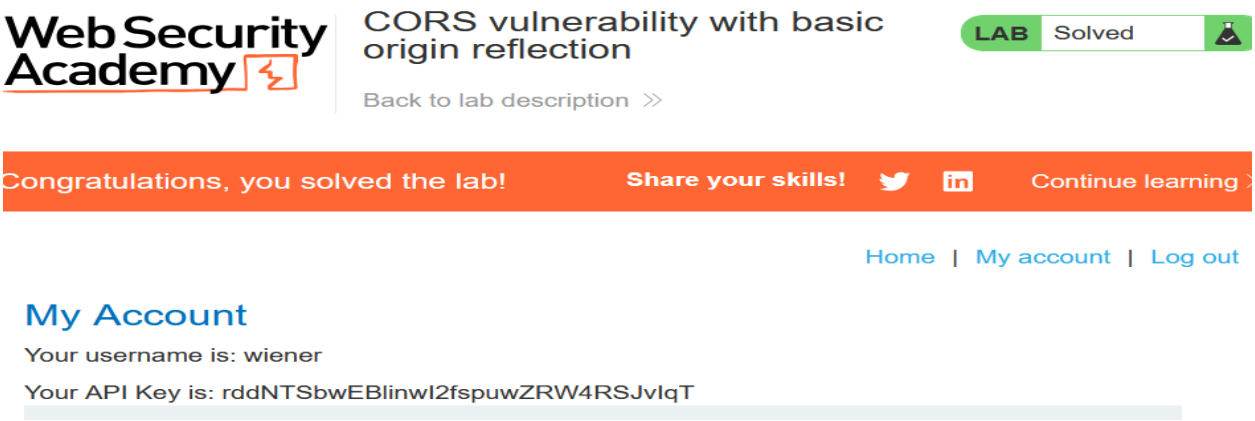
Click View exploit. Observe that the exploit works - you have landed on the log page and your API key is in the URL.

Go back to the exploit server and click Deliver exploit to victim.

Click Access log, retrieve and submit the victim's API key to complete the lab.

```
; rv:133.0) Gecko/20100101 Firefox/133.0"
; rv:133.0) Gecko/20100101 Firefox/133.0"
NT 10.0; Win64; x64; rv:133.0) Gecko/20100101 Firefox/133.0"
33.0) Gecko/20100101 Firefox/133.0"
Win64; x64; rv:133.0) Gecko/20100101 Firefox/133.0"
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36"
22%22,%20%20%22apikey%22:%20%22QfmOAapzwCjHArL4PIwxDQiOPIFBwZTs%22,%20%20%22se
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36"
3.0) Gecko/20100101 Firefox/133.0"
NT 10.0; Win64; x64; rv:133.0) Gecko/20100101 Firefox/133.0"
33.0) Gecko/20100101 Firefox/133.0"
```





HTTP STRICT TRANSPORT SECURITY (HSTS)

Understanding HSTS

HTTP Strict Transport Security (HSTS) is a web security policy mechanism that helps protect websites against protocol downgrade attacks and cookie hijacking. It allows web servers to declare that web browsers (or other complying user agents) should only interact with it using secure HTTPS connections and never via the insecure HTTP protocol.

Let’s understand in simple way;

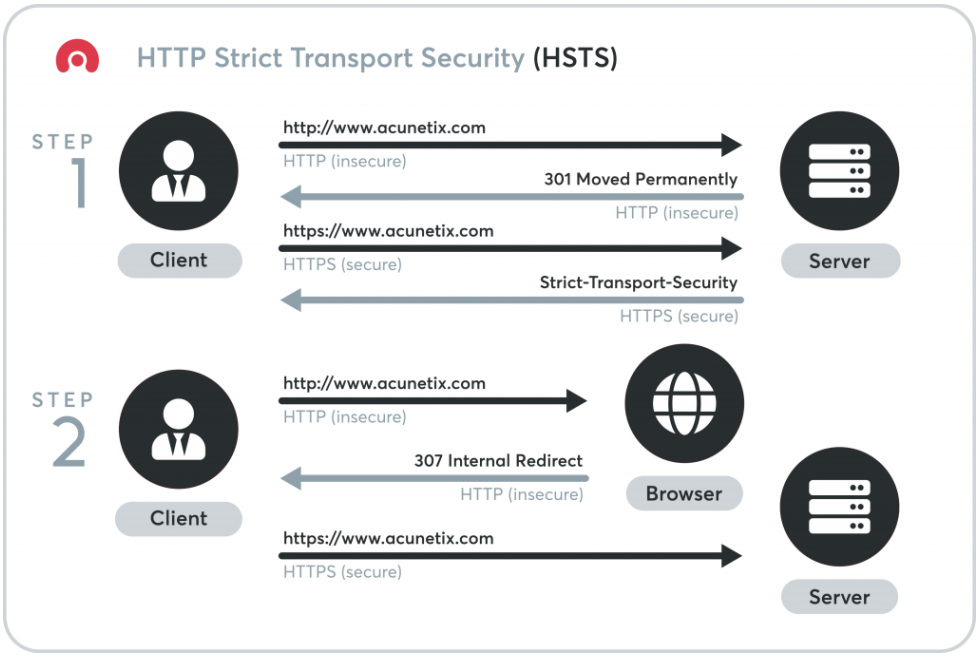
Imagine you're sending a secret message to a friend. You wouldn't want anyone else to read it, right? That's where HSTS comes in.

HSTS is like a secret handshake between your browser and a website. It tells your browser, "Hey, from now on, you can only talk to this website using a super-secure connection (HTTPS)." This prevents anyone from snooping on your messages or changing them along the way.

Here's the simple breakdown:

HTTPS is like a locked box: It keeps your information safe and secret.

HSTS is like a promise: The website promises to always use the locked box, and your browser agrees to use it too.



How HSTS Works ?

When your browser tries to connect to an HSTS-protected site using HTTP, it is redirected to an HTTPS site. Then, the browser receives an HSTS header. From this moment, your browser will remember to only use HTTPS when connecting to this site and will not try HTTP anymore (for a time defined in the HSTS header, usually a year).

Suppose a user visits an HSTS-enabled website, "https://test.com." The website then responds with a “Strict-Transport-Security” header with an HSTS directive, indicating that all future connections should be made over HTTPS. Typically, the header includes the expiration time and specifies the applicable domain. Following is an example of a Strict-Transport-Security header that returns from an HSTS-enabled website.

```
Strict-Transport-Security: max-age = 31536000; includeSubDomains
```

This header information instructs the browser that all subdomains will be HTTPS for one year, blocking subdomains that only support HTTP. The browser then stores this information for the duration mentioned in the header.

When the browser attempts to access that domain in the future, it automatically converts any attempt to access the website via HTTP to HTTPS. This conversion occurs even if the user clicks on an HTTP link within the website or manually types a subdomain without including the protocol part.

In the case of our example HSTS-enabled website, if a subdomain such as “http://test-sub.com” is encountered, the browser will automatically change it to "https://test-sub.com". This conversion occurs without needing to contact the server over HTTP, thereby ensuring secure communication. Once the specified expiration time in the Strict-Transport-Security header has passed, subsequent attempts to load the site via HTTP will resume their regular behavior instead of automatically transitioning to HTTPS. This policy gets renewed each time the browser encounters the header.

Is HSTS Completely Secure ?

Unfortunately, the first time that you access the website, you are not protected by HSTS. If the website adds an HSTS header to an HTTP connection, that header is ignored. This is because an attacker can remove or add headers during a man-in-the-middle attack. The HSTS header cannot be trusted unless it is delivered via HTTPS.

Benefits of HSTS

- Enhanced Security: Protects against common web attacks.
- Improved User Experience: Removes the need for manual HTTPS redirects.
- Prevents Accidental HTTP Requests: Ensures all communication is secure.

Limitations of HSTS

- Initial Connection: HSTS only applies after the initial HTTPS connection is established.
- Configuration Errors: Incorrectly configured HSTS can lead to website inaccessibility.
- Preload List: While helpful, it may not include all HSTS-enabled websites.

Impact of HSTS

- MiTM Attack
- Can be escalated to- Data Theft,
  Unauthorized access,
  Injection of malicious code/payload.
- Cookie Hijacking
- HTTP Downgrade Attacks

How to mitigate HSTS

- Careful Configuration:
  - ✓ max-age Directive: Set an appropriate max-age value. Too short, and security benefits diminish. Too long, and it becomes difficult to recover from configuration errors.
  - ✓ includeSubDomains: Use with caution, as it applies HSTS to all subdomains. Ensure all subdomains are HTTPS-capable.
  - ✓ Preload List: Consider submitting your domain to the preload list for immediate HSTS enforcement.
- Testing and Monitoring:
  - ✓ Thorough Testing: Test HSTS implementation carefully to avoid unintended consequences.
  - ✓ Regular Monitoring: Monitor website logs and browser console for any HSTS-related errors.

- Revocation Mechanism:
  - ✓ preload List Removal: If necessary, remove your domain from the preload list to quickly disable HSTS.
  - ✓ Short max-age: Temporarily reduce the max-age to allow users to access the site via HTTP.

CIATATIONS

1. https://portswigger.net/web-security/cors
2. https://www.skillreactor.io/learn/cross-origin-resource-sharing/types-of-cors-requests
3. https://medium.com/r3d-buck3t/cross-origin-resource-sharing-cors-testing-guide-29616c225a0a#4381
4. https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security.
5. https://www.splunk.com/en_us/blog/learn/hsts-http-strict-transport-security.html

THANK YOU