https://content-security-policy.com/examples/meta/

HTTP response header | comes from server side.

Web Security Policies: Content Security Policy

- Using Content Security Policy a server admin can reduce the chances of XSS by specifying domains that should be considered as valid and trusted sources of executable scripts.
- Content-Security-Policy header is supplied by the server in the HTTP response. The header through a policy directive mentions what are trusted domains for executable scripts.
- For example if an administrator want that scripts from only a specific domain are allowed to be executed. He can mention it in CSP header and supply to the browser

Content-Security-Policy: default-src 'self'; script-src https://www.iitjammu.ac.in

```
Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com; script-src
userscripts.example.com
```

Let's suppose we want to add a CSP policy to our site using the following HTML:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'">
```

Your policy will go inside the `content` attribute of the `meta` tag. The header name `Content-Security-Policy` should go inside the `http-equiv` attribute of the `meta` tag.

The `meta` tag must go inside a `head` tag. The CSP policy only applies to content found after the `meta` tag is processed, so you should keep it towards the top of your document, or at least before any dynamically generated content.

Web Security Policies: Content Security Policy

not stored in browser

```
Content-Security-Policy: default-src 'none'; style-src cdn.example.com; report-uri /_/csp-reports
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sign Up</title>
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    ... Content ...
  </body>
</html>
```

```
"csp-report": {
  "document-uri": "http://example.com/signup.html",
  "referrer": "",
  "blocked-uri": "http://example.com/css/style.css",
  "violated-directive": "style-src cdn.example.com",
  "original-policy": "default-src 'none'; style-src cdn.example.com; report-uri /_/csp-reports"
 }
}
```

Web Security Policies: Content Security Policy

developers.google.com

- `base-uri` restricts the URLs that can appear in a page's `<base>` element.
- `child-src` lists the URLs for workers and embedded frame contents. For example: `child-src https://youtube.com` would enable embedding videos from YouTube but not from other origins.
- `connect-src` limits the origins that you can connect to (via XHR, WebSockets, and EventSource).
- `font-src` specifies the origins that can serve web fonts. Google's web fonts could be enabled via `font-src https://themes.googleusercontent.com`.
- `form-action` lists valid endpoints for submission from `<form>` tags.
- `frame-ancestors` specifies the sources that can embed the current page. This directive applies to `<frame>`, `<iframe>`, `<embed>`, and `<applet>` tags. This directive can't be used in `<meta>` tags and applies only to non-HTML resources.
- `frame-src` was deprecated in level 2, but is restored in level 3. If not present it still falls back to `child-src` as before.
- `img-src` defines the origins from which images can be loaded.
- `media-src` restricts the origins allowed to deliver video and audio.
- `object-src` allows control over Flash and other plugins.
- `plugin-types` limits the kinds of plugins a page may invoke.

The following directives don't use `default-src` as a fallback. Remember that failing to set them is the same as allowing anything.

- `base-uri`
- `form-action`
- `frame-ancestors`
- `plugin-types`
- `report-uri`
- `sandbox`

https://content-security-policy.com/

Web Security Policies: Content Security Policy

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=864000; includeSubDomains
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Referrer-Policy: same-origin
Content-Security-Policy: default-src 'self'; style-src 'self' 'unsafe-inline' https://fonts.googleapis.com http://code.jquery.com http://cdn.ckeditor.com; script-src 'self' 'unsafe-eval' 'unsafe-inline'
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Tue, 21 Jul 2020 10:20:48 GMT
ETag: W/"8d8-17370e3e062"
Content-Type: text/html; charset=UTF-8
Vary: Accept-Encoding
```

```
' 'unsafe-inline' http://cdn.ckeditor.com; font-src 'self' https://fonts.gstatic.com; frame-src https://www.youtube.com/ 'self' http://iitjammu.ac.in; connect-src alpha.iitjammu.ac.in 10.10.10.100 http://
```

```
<img src="img_girl.jpg" alt="Girl in a jacket" width="500" height="600">
```

Web Security Policies: Content Security Policy

developers.google.com https://developers.google.com/web/fundamentals/security/csp/#use_case_3_ssl_only

Inline code is considered harmful

```
<script>
  function doAmazingThings() {
    alert('YOU AM AMAZING!');
  }
</script>
<button onclick='doAmazingThings();'>Am I amazing?</button>
```

to something more like:

```
<!-- amazing.html -->
<script src='amazing.js'></script>
<button id='amazing'>Am I amazing?</button>
```

```
// amazing.js
function doAmazingThings() {
  alert('YOU AM AMAZING!');
}
document.addEventListener('DOMContentLoaded', function () {
  document.getElementById('amazing')
    .addEventListener('click', doAmazingThings);
});
```

Eval too

```
unction() {
  var x = 10;
  var y = 20;
  var a = eval("alert(document.domain)") + "<br>";
  var b = eval("2 + 2") + "<br>";
  var c = eval("x + 17") + "<br>";

  var res = a + b + c;
  document.getElementById("demo").innerHTML = res;
}
</script>
```

↳ web client side computation

↳ if any one writes js in eval then it will run, then it would be dangerous.

Web Security Policies: Subresource Integrity

```
<script src="//www.example.com/script.js" type="text/javascript"></script>;
```

```
<script src="//www.example.com/script.js" type="text/javascript"
integrity="sha256-Abhisa/nS9WMne/YX+dqiFINl+JiE15MCWvASJvVtIk="
crossorigin="anonymous"></script>;
```

"browser try to run" -

window.prompt(eval( ))

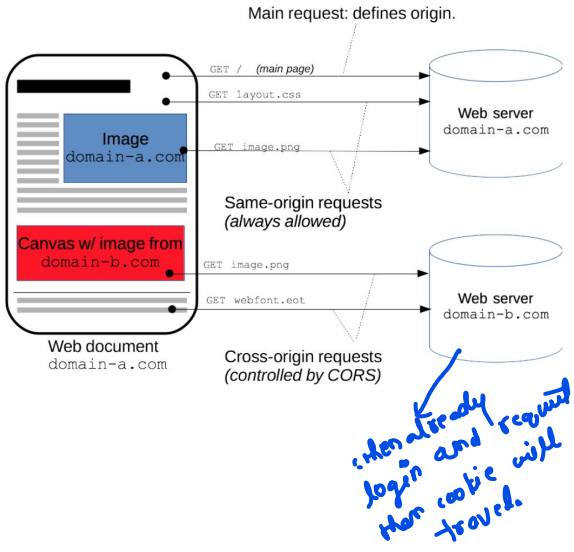
window.open("http..." + window.prompt("?"))

ask for password

Web Security Policies: **Cross Origin Resource Sharing[MDN]**

For example, suppose web content at `https://foo.example` wishes to invoke content on domain `https://bar.other`. Code of this sort might be used in JavaScript deployed on `foo.example`:

```
const xhr = new XMLHttpRequest();
const url = 'https://bar.other/resources/public-data/';

xhr.open('GET', url);
xhr.onreadystatechange = someHandler;
xhr.send();
```



already login and request then cookie will travel.

Web Security Policies: **Cross Origin Resource Sharing[MDN]**

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
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://foo.example
```

↳ can delete using burp suite

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[…XML Data…]
```

The HTTP response headers

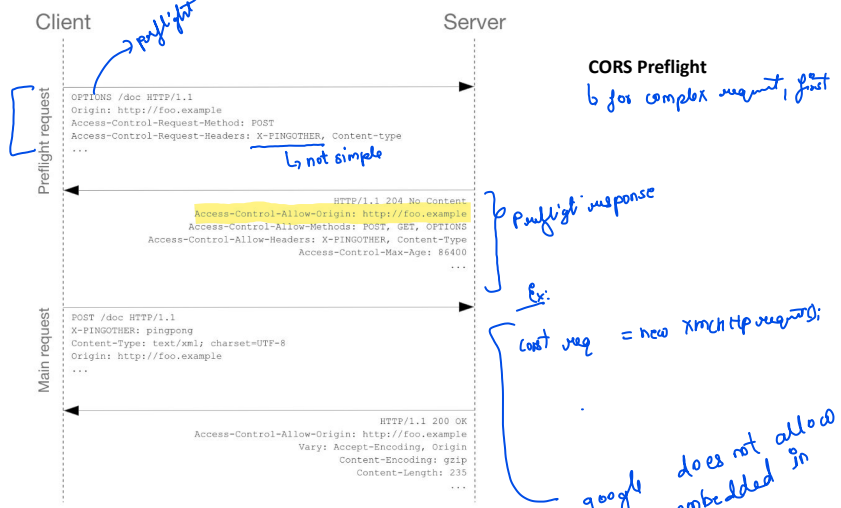
```
Access-Control-Allow-Origin: https://mozilla.org
Vary: Origin
```

```
Access-Control-Max-Age: <delta-seconds>
```

```
Access-Control-Allow-Credentials: true
```

```
Access-Control-Allow-Methods: <method>[, <method>]*
```

Web Security Policies: **Cross Origin Resource Sharing[MDN]**



CORS Preflight

for complex request, first

not simple

Preflight response

Ex:

```
const req = new XMLHttpRequest();
```

google does not allow itself to embedded in other side.

can email using js

Cross Site Scripting

https://excess-xss.com

Doubt ?

Cross-site scripting (XSS) is a code injection attack that allows an attacker to execute malicious JavaScript in another user's browser.

The attacker does not directly target his victim. Instead, he exploits a vulnerability in a website that the victim visits, in order to get the website to deliver the malicious JavaScript for him.

The only way for the attacker to run his malicious JavaScript in the victim's browser is to inject it into one of the pages that the victim downloads from the website.

In the example below, a simple server-side script is used to display the latest comment on a website:

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

The script assumes that a comment consists only of text. However, since the user input is included directly, an attacker could submit this comment: "`<script>...</script>`". Any user visiting the page would now receive the following response:

```
<html>
Latest comment:
<script>...</script>
</html>
```

When the user's browser loads the page, it will execute whatever JavaScript code is contained inside the `<script>` tags. The attacker has now succeeded with his attack.

Cross Site Scripting: XSS

https://excess-xss.com

What is malicious JavaScript?

The possibility of JavaScript being malicious becomes more clear when you consider the following facts:

JavaScript has access to some of the user's sensitive information, such as cookies.

JavaScript can send HTTP requests with arbitrary content to arbitrary destinations by using XMLHttpRequest and other mechanisms.

JavaScript can make arbitrary modifications to the HTML of the current page by using DOM manipulation methods.

```
document.title="Phishing"

window.location = "https://iitjammu.ac.in"
```

automatically redirect to this

can cause phishing

Keylogger script for browser

```
var keys = '';
document.onkeypress = function(e) {
  get = window.event ? event : e;
  key = get.keyCode ? get.keyCode : get.charCode;
  key = String.fromCharCode(key);
  keys += key;
  window.prompt(key);
}
```

The consequences of malicious JavaScript

Among many other things, the ability to execute arbitrary JavaScript in another user's browser allows an attacker to perform the following types of attacks:

Cookie theft: The attacker can access the victim's cookies associated with the website using `document.cookie`, send them to his own server, and use them to extract sensitive information like session IDs.

Keylogging: The attacker can register a keyboard event listener using `addEventListener` and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.

Phishing: The attacker can insert a fake login form into the page using DOM manipulation, set the form's `action` attribute to target his own server, and then trick the user into submitting sensitive information.

Cross Site Scripting: XSS

https://excess-xss.com

What is malicious JavaScript?

This fact highlights a key issue:

> *If an attacker can use your website to execute arbitrary JavaScript in another user's browser, the security of your website and its users has been compromised.*

Actors in an XSS attack

Before we describe in detail how an XSS attack works, we need to define the actors involved in an XSS attack. In general, an XSS attack involves three actors: **the website**, **the victim**, and **the attacker**.

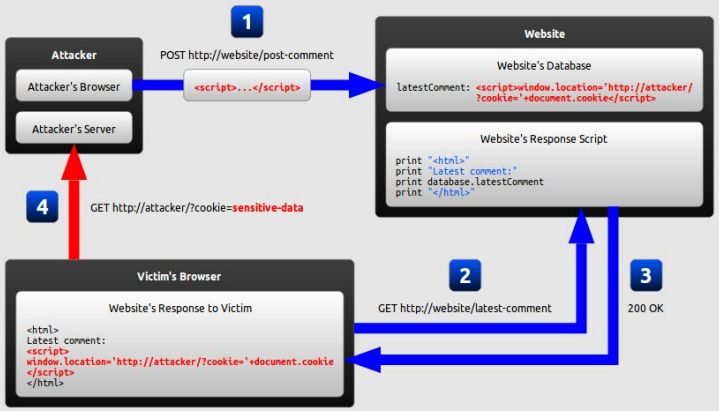
- **The website** serves HTML pages to users who request them. In our examples, it is located at `http://website/`.
  - **The website's database** is a database that stores some of the user input included in the website's pages.
- **The victim** is a normal user of the website who requests pages from it using his browser.
- **The attacker** is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.
  - **The attacker's server** is a web server controlled by the attacker for the sole purpose of stealing the victim's sensitive information. In our examples, it is located at `http://attacker/`.

Cross Site Scripting: XSS

https://excess-xss.com

How the example attack works

The diagram below illustrates how this example attack can be performed by an attacker:



Cross Site Scripting: XSS

https://excess-xss.com

Types of XSS

While the goal of an XSS attack is always to execute malicious JavaScript in the victim's browser, there are few fundamentally different ways of achieving that goal. XSS attacks are often divided into three types:

- **Persistent XSS**, where the malicious string originates from the website's database.
- **Reflected XSS**, where the malicious string originates from the victim's request.
- **DOM-based XSS**, where the vulnerability is in the client-side code rather than the server-side code.

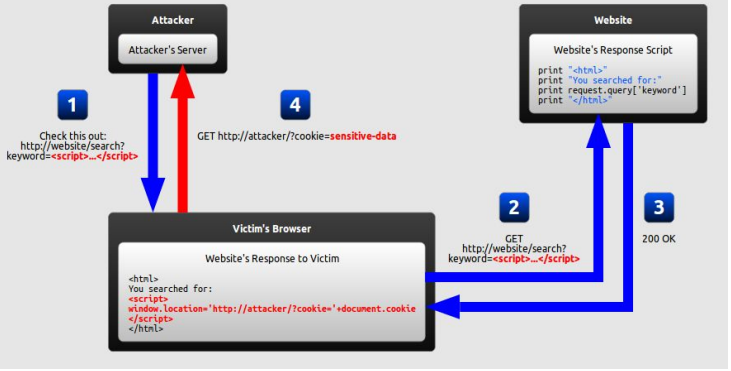
The previous example illustrated a persistent XSS attack. We will now describe the other two types of XSS attacks: reflected XSS and DOM-based XSS.

Cross Site Scripting: XSS

https://excess-xss.com

Reflected XSS

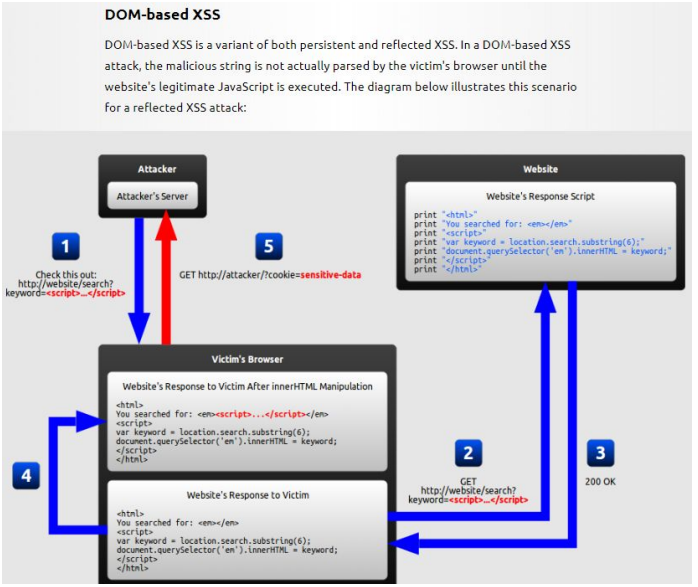
In a reflected XSS attack, the malicious string is part of the victim's request to the website. The website then includes this malicious string in the response sent back to the user. The diagram below illustrates this scenario:



Cross Site Scripting: DOM based XSS

DOM-based XSS

DOM-based XSS is a variant of both persistent and reflected XSS. In a DOM-based XSS attack, the malicious string is not actually parsed by the victim's browser until the website's legitimate JavaScript is executed. The diagram below illustrates this scenario for a reflected XSS attack:



Cross Site Scripting: XSS DOM based XSS

acunetix

```
<html>
<head>
<title>Custom Dashboard </title>
...
</head>
Main Dashboard for
<script>
        var pos=document.URL.indexOf("context=")+8;
        document.write(document.URL.substring(pos,document.URL.length));
</script>
...
</html>
```

http://www.example.com/userdashboard.html#context=alert(document.cookie)

XSS Solutions

Convert some characters to HTML entities:

```
<?php
$str = '<a href="https://www.w3schools.com">Go to w3schools.com</a>';
echo htmlentities($str);
?>
```

The HTML output of the code above will be (View Source):

```
&lt;a href=&quot;https://www.w3schools.com&quot;&gt;Go to w3schools.com&lt;/a&gt;
```

```
<!DOCTYPE html>
<html>
<body>

<h1>HTML Entity Example</h1>

<h2>The greater-than sign: &gt;</h2>

</body>
</html>
```

HTML Entity Example

The greater-than sign: >

XSS Solutions

```
<script>alert(document.domain)</script>
```

```
&#x3C;script&#x3E;alert(document.domain)&#x3C;/script&#x3E;
```

Result	Description	Entity name	Entity number
	Non-breaking space		
<	Less than	&lt;	&#60;
>	Greater than	&gt;	&#62;
&	Ampersand	&amp;	&#38;

Methods of preventing XSS

Recall that an XSS attack is a type of code injection: user input is mistakenly interpreted as malicious program code. In order to prevent this type of code injection, secure input handling is needed. For a web developer, there are two fundamentally different ways of performing secure input handling:

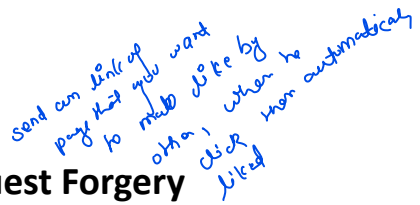
- **Encoding**, which escapes the user input so that the browser interprets it only as data, not as code.
- **Validation**, which filters the user input so that the browser interprets it as code without malicious commands.

IBM Developer

To help prevent XSS attacks, an application needs to ensure that all variable output in a page is encoded before being returned to the end user. Encoding variable output substitutes HTML markup with alternate representations called *entities*. The browser displays the entities but does not run them. For example, `<script>` gets converted to `&lt;script&gt;`.

When a web browser encounters the entities, they will be converted back to HTML and printed but they will not be run. For example, if an attacker injects `<script>alert("you are attacked")</script>` into a variable field of a server's web page, the server will, using this strategy, return `&lt;script&gt;alert("you are attacked")&lt;/script&gt;`.

When the web browser downloads the encoded script, it will convert the encoded script back to `<script>alert("you are attacked")</script>` and display the script as part of the web page but the browser will not run the script.

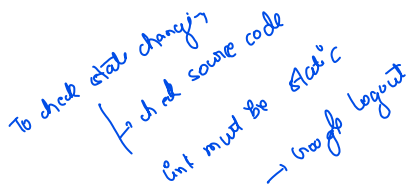


Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth.



Cross Site Request Forgery

•How does the attack work ? HTTP GET Scenario

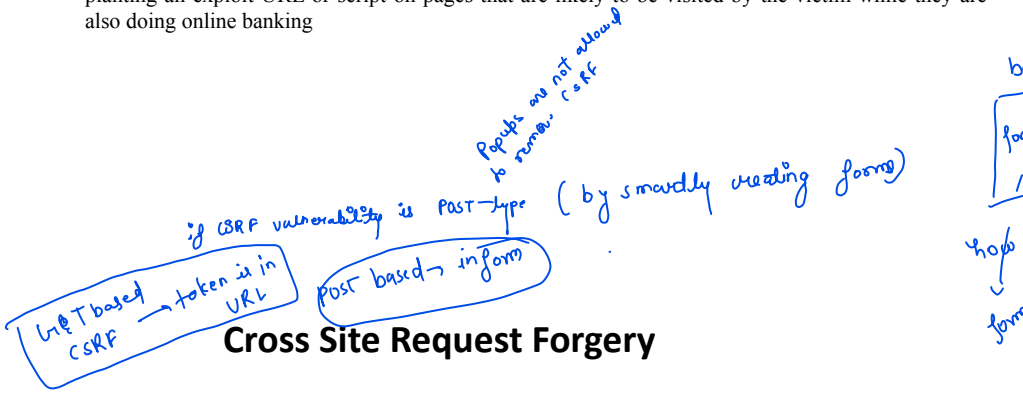
- Let us consider the following example: Alice wishes to transfer $100 to Bob using the bank.com web application that is vulnerable to CSRF.

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

- Maria now decides to exploit this web application vulnerability using Alice as her victim. Maria first constructs the following exploit URL which will transfer $100,000 from Alice's account to her account.

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

- The social engineering aspect of the attack tricks Alice into loading this URL when she's logged into the bank application. This is usually done with one of the following techniques:
  - sending an unsolicited email with HTML content
  - planting an exploit URL or script on pages that are likely to be visited by the victim while they are also doing online banking



Cross Site Request Forgery

The exploit URL can be disguised as an ordinary link, encouraging the victim to click it:

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>
```

Or as a 0x0 fake image:

```
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">
```

If this image tag were included in the email, Alice wouldn't see anything. However, the browser *will still* submit the request to bank.com without any visual indication that the transfer has taken place.

•How does the attack work ? HTTP POST Scenario

- The only difference between GET and POST attacks is how the attack is being executed by the victim. Let's assume the bank now uses POST and the vulnerable request looks like this:

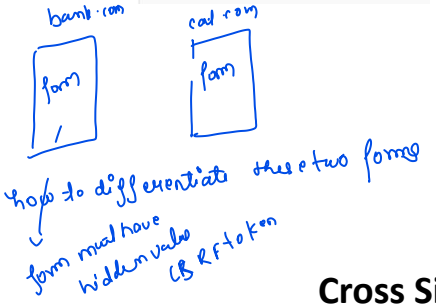
```
POST http://bank.com/transfer.do HTTP/1.1

acct=BOB&amount=100
```

Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tag:

```
<form action="<nowiki>http://bank.com/transfer.do</nowiki>" method="POST">
<input type="hidden" name="acct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="View my pictures"/>
</form>
```

This form will require the user to click on the submit button, but this can be also executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form...
```



Cross Site Request Forgery


```
<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />
```

•Preventive Measures ?

- Add a per-request nonce to the URL and all forms in addition to the standard session. This is also referred to as "form keys".
- Add a hash (session id, function name, server-side secret) to all forms.
- Checking the referrer header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function.
- Secure Cookies: SameSite: Strict
- "Although CSRF is fundamentally a problem with the web application, not the user, users can help protect their accounts at poorly designed sites by logging off the site before visiting another, or clearing their browser's cookies at the end of each browser session.

Cross Site Request Forgery


```
<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />
```

Anti-CSRF Tokens

The most popular implementation to prevent Cross-site Request Forgery (CSRF), is to make use of a challenge token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a *CSRF Token* or a *Synchronizer Token*, works as follows:

- The web server generates a token
- The token is statically set as a hidden input on the protected form
- The form is submitted by the user
- The token is included in the POST data
- The web application compares the token generated by the web application with the token sent in through the request
- If these tokens match, the request is valid, as it has been sent through the actual form in the web application
- If there is no match, the request will be considered as illegal and will be rejected.

This protects the form against Cross-site Request Forgery (CSRF) attacks, because an attacker crafting a request will also need to guess the anti-CSRF token for them to successfully trick a victim into sending a valid request. What's more, is that this token should be invalidated after some time and after the user logs out.

https://www.acunetix.com/websitesecurity/csrf-attacks/

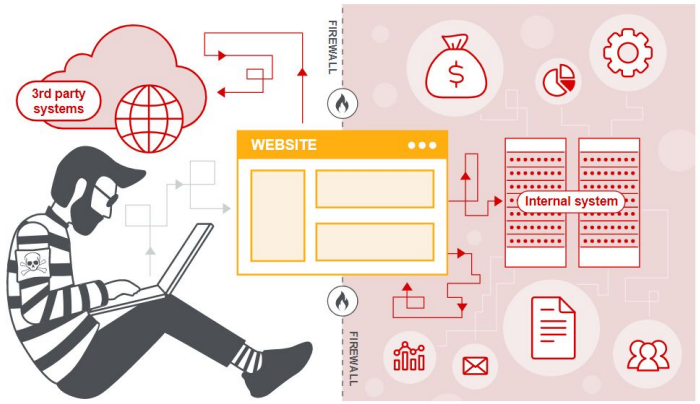
Same Site Cookies

CSRF attacks are only possible since Cookies are always sent with any requests that are sent to a particular origin, which is related to that Cookie. Due to the nature of a CSRF attack, a flag can be set against a Cookie, tuning it into a same-site Cookie. A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. The Cookie and the page from where the request is being made, are considered to have the same origin if the protocol, port (if applicable) and host is the same for both.

What is SSRF?

Server-side request forgery (also known as SSRF) is a web security vulnerability that allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems, potentially leaking sensitive data such as authorization credentials.



https://portswigger.net/

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3
D6%26storeId%3D1
```

This causes the server to make a request to the specified URL, retrieve the stock status, and return this to the user.

In this situation, an attacker can modify the request to specify a URL local to the server itself. For example:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

Here, the server will fetch the contents of the `/admin` URL and return it to the user.

In the preceding example, suppose there is an administrative interface at the back-end URL `https://192.168.0.68/admin`. Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://192.168.0.68/admin
```

https://portswigger.net/

Circumventing Common SSRF Defenses

- Alternate IP representation- Convert IP to Hex or Octal format
- Fake domain resolution- Get your own domain resolve to a local host IP 127.0.0.1
- If a whitelisting is there you can use these measures
  - http://abc.com@bcz.com the actual request is to bcz.com
- You can prevent it with whitelist and blacklists of requests from web server to internal systems
- Proper response handling so that responses from Internal systems should not be provided to an untrusted client as is
- Authentication on Internal Servers

Dr. Gaurav Varshney, IIT Jammu