

# Attacking Users: Other Techniques

# Attacking Users: Other Techniques

- XSS is the "grandfather" of attacks on users.
- However, other advanced techniques exist that can bypass XSS defenses.
- These attacks are often more complex than Cross-Site Scripting (XSS) and can work even when XSS is not possible.
- One such attack is **Request Forgery**

# Request Forgery

- In request forgery, an attacker tricks a user's browser into performing actions that the user didn't intend.
- Also called session riding
- Related to session hijacking
- Request forgery doesn't require knowing the session token. Instead, it exploits how web browsers handle requests.

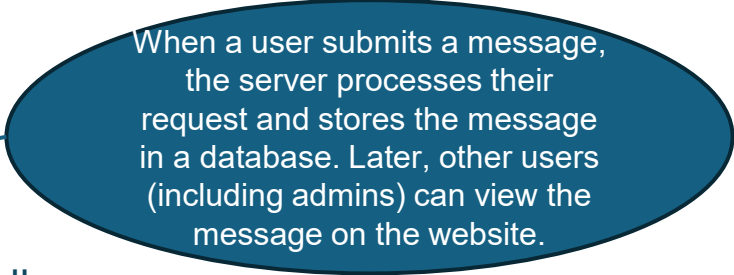
# Types of Request Forgery

- 1. **On-Site Request Forgery (OSRF)**: Happens within the same application (e.g., example.com tricks users on example.com).
- 1. **Cross-Site Request Forgery (CSRF)**: Happens between different sites (e.g., malicioussite.com tricks users into performing actions on example.com).

# On-Site Request Forgery (OSRF)

- Used with stored XSS vulnerabilities
- Imagine a message board where users can submit messages.

POST /submit.php  
Host: wagh-app.com  
Content-Length: 34



When a user submits a message, the server processes their request and stores the message in a database. Later, other users (including admins) can view the message on the website.

type=question&name=John&message=Hello

The server takes this input and displays the message like this on the page

```
<tr>
  <td></td>
  <td>John</td>
  <td>Hello</td>
</tr>
```

# On-Site Request Forgery (OSRF)

## What the Attacker Does

An attacker notices that they can control part of the type parameter (the URL in the <img> tag). This means the attacker can make any user viewing their message unknowingly send a GET request to a URL of the attacker's choice.

*POST /submit.php  
Host: wahh-app.com  
Content-Length: 100*

*type=../admin/newUser.php?username=hacker&password=12345&role=admin#&name=Attacker&message=Hacked*

When someone views the attacker's message, the server generates this HTML

# On-Site Request Forgery (OSRF)

For **normal users**: The request fails because they don't have admin privileges.

For **admin users**: When they view the message, their session token (with admin privileges) is used to send the GET request.

The server processes the request as if the **admin** is asking to create a new user named hacker with admin privileges.

Now, the attacker has created a new admin account (hacker with password 12345) without the real admin realizing it.

# How Could This Be Prevented?

- Validate the **type** parameter to only allow predefined values like question or answer.
- Block suspicious characters in input (/ , . , ? , & , =) if they are not needed.
- Avoid trusting user input directly when creating HTML output.



# Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack where the attacker tricks a user into performing actions they didn't intend on a web application, usually one where the user is already logged in. This can happen when a website is vulnerable and doesn't properly protect against such attacks.

# Cross-Site Request Forgery (CSRF)

- CSRF uses the fact that web browsers automatically send cookies associated with a domain when making a request. This means that if you're logged into a website, your browser sends the login session (stored as cookies) with every request to that site.
- In a CSRF attack, the attacker creates a page that looks harmless (like a normal website). When a user visits this page, their browser automatically submits a request to the target website (where the user is logged in) and includes their session cookies.
- The target website processes the request as if it came directly from the legitimate user, allowing the attacker to perform actions (like changing account settings or making purchases) without the user's knowledge.

# Same-Origin Policy

- Does not prevent a website from issuing requests to a different domain
- Prohibits the originating website from processing the responses from other domains
- CSRF attacks are "one-way"
- Multistage attacks are not possible with a pure CSRF attack

# Example

Consider a website where admins can create new user accounts by submitting the following form

*POST /auth/390/NewUserStep2.ashx HTTP/1.1*


*Host: mdsec.net*

*Cookie: SessionId=8299BE6B260193DA076383A2385B07B9*

*Content-Type: application/x-www-form-urlencoded*

*Content-Length: 83*

*realname=daf&username=daf&userrole=admin&password=letmein1&confirmpassword=letmein1*



Three Features  
that Make It  
Vulnerable

1. The request **creates a new user** with **admin privileges**.
2. The website relies **only on cookies** to track the user's session (no special tokens or headers).
3. The attacker knows **all the parameters** needed to submit the form (like the username, password, and role).

Because of this, the attacker can create a page that, when visited by the victim, **automatically submits the request** to create a new admin account.

# Example

The attacker builds a simple web page with the following HTML code

```
<html>
<body>
<form action="https://mdsec.net/auth/390/NewUserStep2.ashx" method="POST">
  <input type="hidden" name="realname" value="daf">
  <input type="hidden" name="username" value="daf">
  <input type="hidden" name="userrole" value="admin">
  <input type="hidden" name="password" value="letmein1">
  <input type="hidden" name="confirmpassword" value="letmein1">
</form>
<script>
  document.forms[0].submit();
</script>
</body>
</html>
```

The page contains a hidden form with all the data required to create the new admin account.

Since the browser automatically includes the session cookies, the request is processed as if it came from the logged-in user. If the user is an admin, the new admin account is created.

# Defend Against CSRF

To prevent CSRF attacks, websites can use the following defenses:

- 1.Anti-CSRF Tokens:** Include a unique token in every form. This token is checked on the server to ensure the request is legitimate.
- 2.SameSite Cookies:** Use the SameSite cookie attribute to prevent cookies from being sent along with cross-origin requests,e.g lax and strict .
- 3.Referer Header Check:** Ensure requests are coming from the same domain by checking the Referer header.

# Exploiting CSRF Flaws

- Most common flaw: Application relies solely on HTTP cookies for tracking sessions
- Browser automatically sends cookie with every request

# Authentication and CSRF

## Real-World Example

### CSRF in Home DSL Routers

Many **home DSL routers** have web interfaces that allow users to configure settings, such as **opening ports on the firewall**. This is a powerful and potentially dangerous action because it can expose the internal network to external threats. However, many of these devices are vulnerable to CSRF because:

- The router's web interface **may not protect sensitive actions against CSRF**.
- **Users often don't change the default login credentials**, which makes these devices even more vulnerable to attacks.



# How the Attack Works in This Case:

1. **Default credentials** are commonly used for the router login (e.g., username: admin, password: admin).
2. The attacker creates a **malicious web page** that first submits a **login request** using the default credentials.
3. The router's web interface **logs in the victim** and sets a **session cookie** in their browser.
4. The attacker's page then submits a **CSRF request** to the router's web interface to perform a sensitive action, such as opening ports on the firewall.
5. The request is **processed with the victim's session** (since they are logged in), allowing the attacker to carry out the malicious action.

# Preventing CSRF Flaws

- Supplement HTTP cookies with additional methods of tracking sessions
- Typically, hidden fields in HTML forms
- This blocks CSRF attacks, if the attacker has no way to determine the value of the "anti-CSRF token"
- Tokens must not be predictable, and must be tied to a session so they can't be re-used from a different session

# How XSS Can Bypass Anti-CSRF Defenses

## 1. Anti-CSRF Defenses

1. Anti-CSRF tokens protect sensitive requests by ensuring actions come from legitimate sources.
2. XSS (Cross-Site Scripting) vulnerabilities can sometimes bypass these protections.

## 2. XSS & Anti-CSRF Interaction

1. **XSS** allows attackers to execute JavaScript on victims' browsers.
2. The attacker's script can read **anti-CSRF tokens** in the response and submit them in subsequent requests.

## 3. Challenges to XSS Exploiting Anti-CSRF

1. **Reflected XSS** requires the attacker to already have the anti-CSRF token.
2. **Stored XSS** allows attackers to steal tokens directly from the page.

# Example

- website, bank.com, uses anti-CSRF tokens to protect sensitive actions like transferring money. The anti-CSRF token is included in a form for transferring money

```
<form action="/transfer" method="POST">
```

```
<input type="hidden" name="csrf_token" value="abc123">
```

```
<input type="text" name="amount" placeholder="Enter amount">
```

```
<input type="submit" value="Transfer">
```

```
</form>
```

bank.com has an XSS vulnerability that allows attackers to inject malicious JavaScript into the site.

# What attacker did

```
<script>
```

```
// Malicious script to steal the anti-CSRF token
```

```
var csrfToken = document.querySelector('input[name="csrf_token"]').value;
```

```
// Use the token to send a forged request
```

```
fetch('https://bank.com/transfer', {
```

```
  method: 'POST',
```

```
  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
```

```
  body: 'csrf_token=' + csrfToken + '&amount=10000'
```

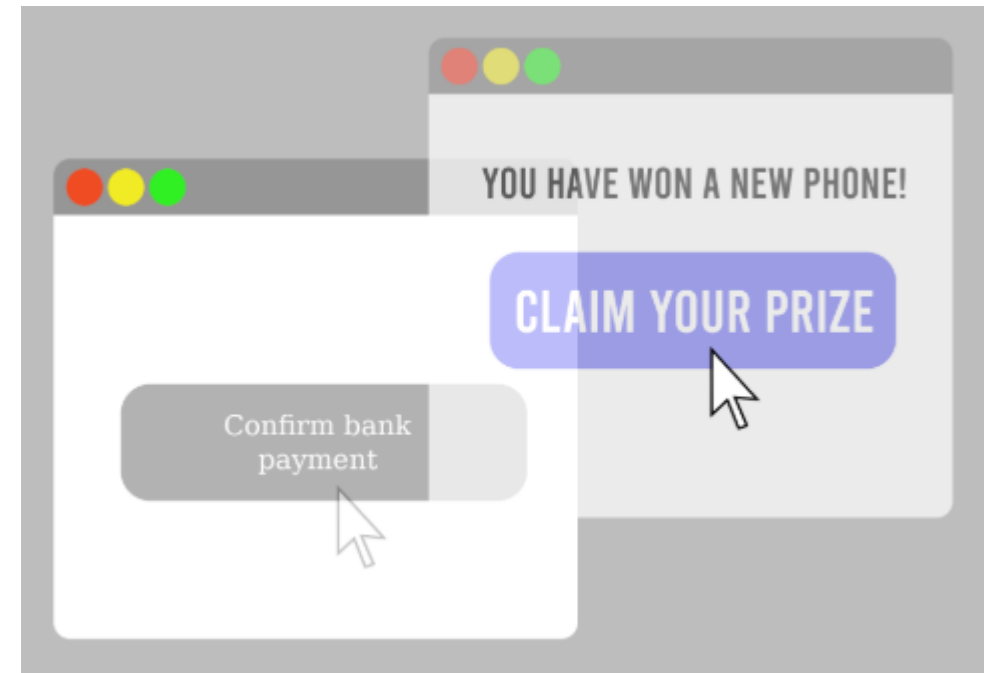
```
});
```

```
</script>
```

# UI Redress (Clickjacking)

- UI Redress is a type of attack where a third-party website tricks a user into performing unintended actions on another website.

Even if anti-CSRF tokens are used, UI redress attacks can still succeed because the requests come from the target application, not a third-party site.



# Iframe Attack

- The attacker embeds the target application in a transparent iframe on their page, overlaying it with their own interface.
- When the user interacts with the attacker's interface (e.g., clicking a button), they are actually interacting with the target application behind the scenes.

## **EXAMPLE**

A banking website requires the user to submit payment details and then click a "Confirm" button.

The attacker creates a webpage with an iframe containing the target bank's page.

The attacker's page overlays their own button or interface on top of the "Confirm" button.

The user thinks they are clicking the attacker's button but, in reality, they are confirming a payment on the bank's site, transferring money unknowingly.

# Iframe Attack

```
<html>  
  <head>  
    <title>Click here for a gift!</title>  
  </head>  
  <body>  
    <button>Click here for a gift!</button>
```

```
    <iframe src="https://target-website.com/confirm-payment"  
      style="opacity: 0; position: absolute; width: 100%; height: 100%; top: 0; left: 0;">  
    </iframe>  
  </body>  
</html>
```



# Advanced UI Redress Attack Techniques

## 1. Inducing Text Input:

1. The attacker can create a form that encourages the user to type something (like entering a phone number).
2. A script can then send the typed characters to the target page's input fields, without the user knowing.

**Example:** If the attacker's form asks for a "phone number," while the user types, the script places the input in a **funds transfer amount field** on the bank's site.

## 1. Inducing Mouse Dragging:

2. The attacker can also trick the user into performing mouse actions like dragging items (text, links) into form fields on the attacker's page, which can capture sensitive data.

**Example:** Dragging an email address into a form could unknowingly create a rule to forward all emails to the attacker's inbox.

# Framebusting Defenses

- Framebusting is a defense technique used by websites to prevent their pages from being loaded inside iframes on other sites, thus protecting against attacks like UI redress (clickjacking).
- It involves using a script on the page that checks if the page is loaded inside an iframe. If it is, the script attempts to break out of the iframe or takes some defensive action, like redirecting the user to an error page.

# X-Frame-Options HTTP Header

The X-Frame-Options header instructs browsers whether a page can be embedded in a frame.

**DENY:** Prevents the page from being framed by any site.

**SAMEORIGIN:** Allows framing only if the framing page is from the same origin.

**ALLOW-FROM <URI>** (Deprecated): Allows framing only by a specific site. This is now largely replaced by Content Security Policy.

## (CSP) Frame-Ancestors Directive

### Content Security Policy

The frame-ancestors directive in CSP is the modern, more flexible alternative to X-Frame-Options. It specifies which origins are allowed to embed the page in a frame.

### Example:

Content-Security-Policy: frame-ancestors 'self' https://trusted-site.com;

'self': Only allows framing by the same origin.

https://trusted-site.com: Allows framing by a specific site.

# older javascript technique

```
<script>
```

```
  if (top.location != self.location) {  
    top.location = self.location;  
  }
```

```
</script>
```

**top.location:** Refers to the top-level frame (the browser window/tab).

**self.location:** Refers to the current page (which could be an iframe if it's embedded).

# Capturing Data Cross-Domain

# Same origin policy

- Same-origin policy is a security measure in web browsers that prevents one domain from accessing the data from another domain.
- This policy is meant to protect users from malicious activities, such as stealing sensitive information from a different website.
- However, there are ways attackers can bypass this policy to capture data across domains.

# What Defines an "Origin"?

- An origin is defined by the protocol, domain, and port of a URL. Two resources have the same origin only if all three match.
- `https://example.com:443/page1`  
`https://example.com:443/page2` → Same origin.

`https://example.com` and  
`http://example.com` → Different origin (protocol mismatch).

`https://example.com:443` and  
`https://example.com:8080` → Different origin (port mismatch).



# What SOP Restricts

- **DOM Access:** A script from one origin cannot read or modify the DOM of a document from another origin.

In other words, if you have two websites open in separate tabs or if one website tries to access the DOM of another website embedded in an iframe, SOP prevents this for security reasons.

siteA.com tries to access the DOM of siteB.com in an iframe.

```
<!-- siteA.com -->
```

```
<iframe src="https://siteB.com"></iframe>
```

```
<script>
```

```
// This will throw an error due to SOP restriction
```

```
var iframe = document.querySelector('iframe');
```

```
console.log(iframe.contentDocument.body.innerHTML); // Access denied
```

```
</script>
```

# What SOP Restricts

- **Cookies:** The SOP restricts access to cookies to the origin that set them, meaning that a website cannot access or manipulate cookies from another domain unless explicitly allowed.

// SiteA sets a cookie

```
document.cookie = "sessionId=12345; path=/; domain=siteA.com; Secure; HttpOnly; SameSite=Lax";
```

// SiteB tries to read cookies set by siteA

```
console.log(document.cookie); // Output will be empty or restricted
```

# What SOP Restricts

- **AJAX Requests**

AJAX (Asynchronous JavaScript and XML) requests are commonly used to request data from servers without refreshing the page. SOP prevents an AJAX request made from one domain to another domain (cross-origin requests) unless the server explicitly allows it using CORS (Cross-Origin Resource Sharing) headers.

# Techniques for Capturing Data Across Domains

Capturing data across domains typically involves **bypassing browser-enforced security mechanisms** like the **Same-Origin Policy (SOP)**.

# Cross-Origin Resource Sharing (CORS)

## Cross-Origin Resource Sharing (CORS)

CORS is a security feature that allows servers to specify which origins can access their resources. Misconfigurations in CORS can lead to data exposure.

By default, web browsers enforce the Same-Origin Policy (SOP), which restricts web pages from making requests to domains other than their own.

# (CORS)

**Access-Control-Allow-Origin:** This header tells the browser which origins are allowed to access the resources. If this header is missing or misconfigured, the browser will block the request.

Access-Control-Allow-Origin: <https://trusted-site.com>

# (CORS)

If a server sets the `Access-Control-Allow-Origin` header to `*` or reflects the `Origin` header without validation, it opens the door for attackers to make cross-origin requests and read sensitive data.

`Access-Control-Allow-Credentials`: Indicates whether the request can include credentials like cookies, HTTP authentication, or client-side SSL certificates.

`Access-Control-Allow-Origin: *`

`Access-Control-Allow-Credentials: true`

# Assignment 3