The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web

Soheil Khodayari[*], Thomas Barber[†], and Giancarlo Pellegrino[*]

[*]CISPA - Helmholtz Center for Information Security

[†]SAP Security Research

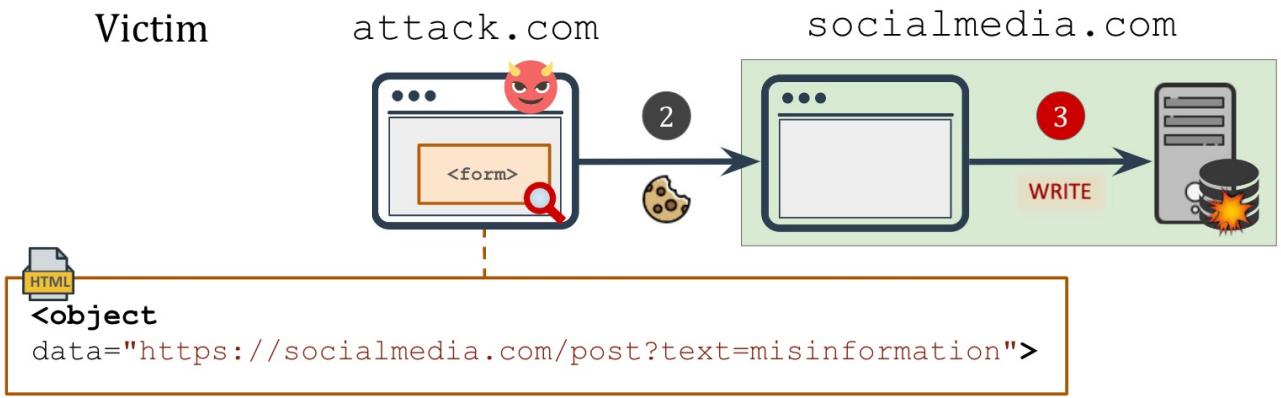
45th IEEE Symposium on Security and Privacy
May 20-23, 2024

SCAN ME

@Soheil__K soheil.khodayari@cispa.de

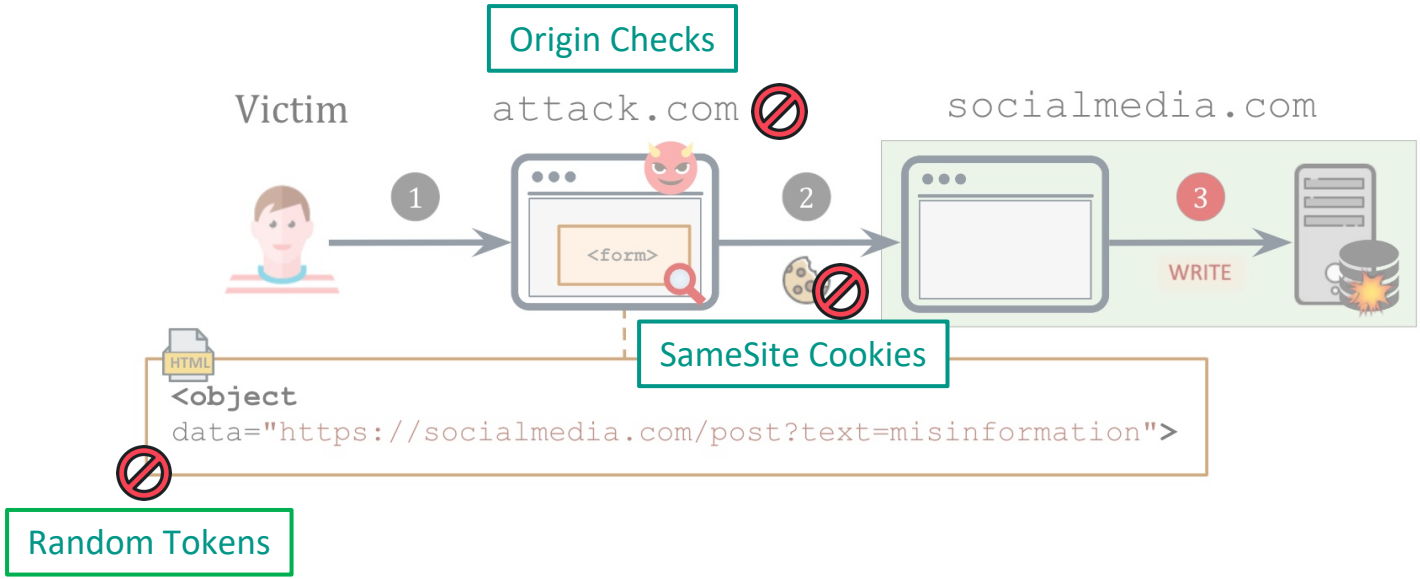
Cross-Site Request Forgery (CSRF)

- Trick user browser to send an authenticated request causing a persistent state change
  - **Root Cause:** server cannot distinguish unintentional from intentional requests



Cross-Site Request Forgery (CSRF)

- Trick user browser to send an authenticated request causing a persistent state change
  - **Root Cause:** server cannot distinguish unintentional from intentional requests
  - Robust defenses well-known ✓



Cross-Site Request Forgery (CSRF)

- Trick user browser to send an authenticated request causing a persistent state change
  - **Root Cause:** server cannot distinguish unintentional from intentional requests
  - Robust defenses well-known

⚠ Did we **solve** CSRF attacks with these defenses ?

Origin Checks

Victim attack.com socia

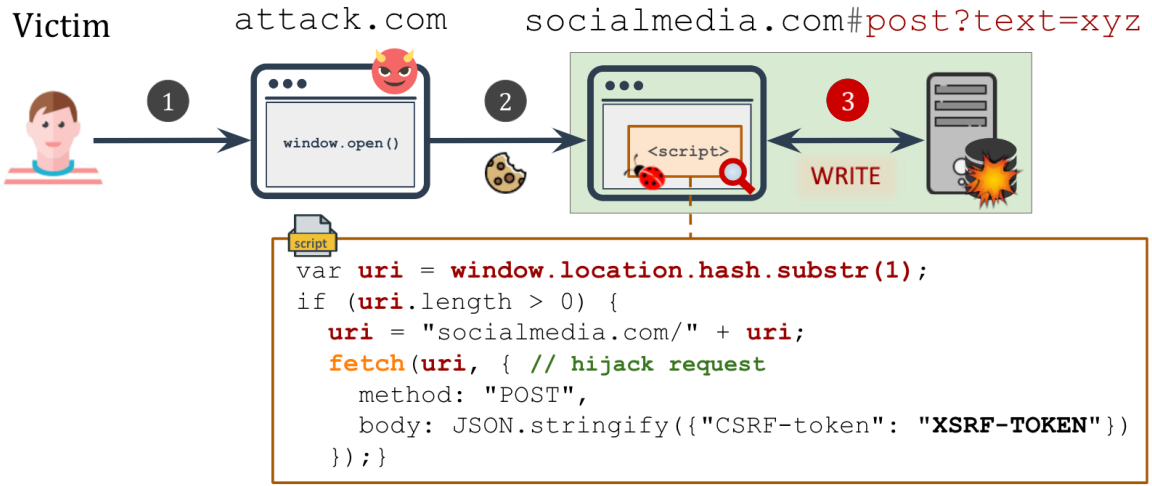
SameSite Cookies

<object
data="https://socialmedia.com/post?text=misinformation">

Random Tokens

Client-side CSRF

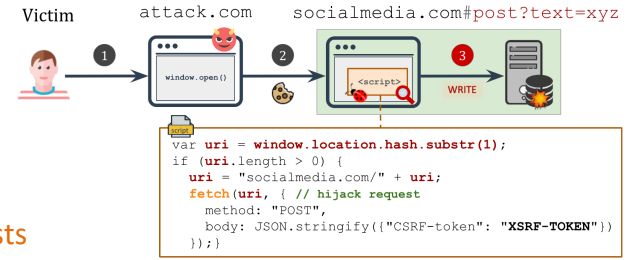
- Exploit input validation vulnerabilities in JavaScript programs to hijack async requests
  - Similar vulnerability affected Instagram in 2018[1]



[1] **Source:** https://www.facebook.com/notes/996734990846339

Problem Statement

- Client-side CSRF only one instance of the larger issue of request hijacking
  - Studied client-side CSRF before [USEC'21]
  - Focused on `XMLHttpRequest` and `Fetch` APIs



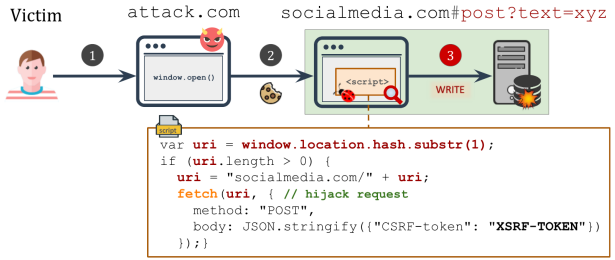
- Other types of HTTP requests and APIs exists
  - The `sendBacon` API accounting for > 35% of the API calls for async requests
  - Web sockets, SSE connections, push notifications, etc

Problem Statement

- Client-side CSRF only one instance of the larger issue of request hijacking
  - Studied client-side CSRF before [USEC'21]
  - Focused

RQ1: Browser APIs and Attacks

- Other types of HTTP requests and APIs exists
  - The `sendBacon` API accounting for > 35% of the API calls for async reque
  - Web sockets, SSE connections, push notifications, etc

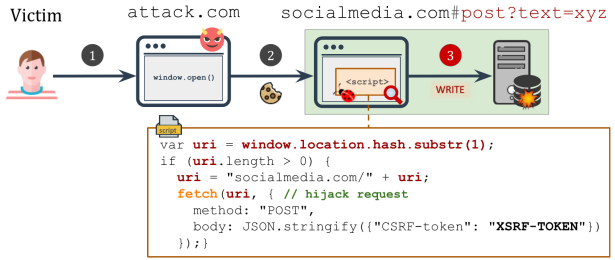


Problem Statement

- Client-side CSRF only one instance of the larger issue of request hijacking
  - Studied client-side CSRF before [USEC'21]
  - Focused

RQ1: Browser APIs and Attacks

- Other types of HTTP requests and APIs exists
  - The `sendBacon` API accounting for > 35% of the API calls for async reque
  - Web sockets, SSE connections, push notifications, etc



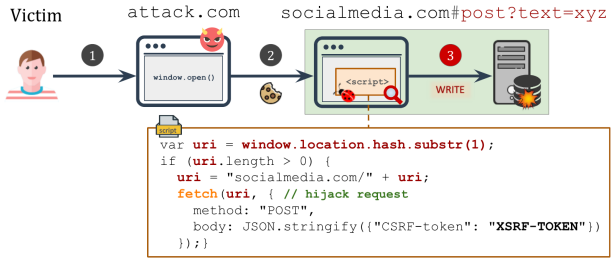
- Attack surface
  - No web measurement available, in-the-wild prevalence of request hijacking unknown

Problem Statement

- Client-side CSRF only one instance of the larger issue of request hijacking
  - Studied client-side CSRF before [USEC'21]
  - Focused
- Other types of HTTP requests and APIs exists
  - The `sendBacon` API accounting for > 35% of the API calls for async reque
  - Web sockets, SSE connections, push notifications, etc
- Attack surfa
  - No web measurement available, in-the-wild prevalence of request hijacking unknown

RQ1: Browser APIs and Attacks

RQ2: Detection and Prevalence



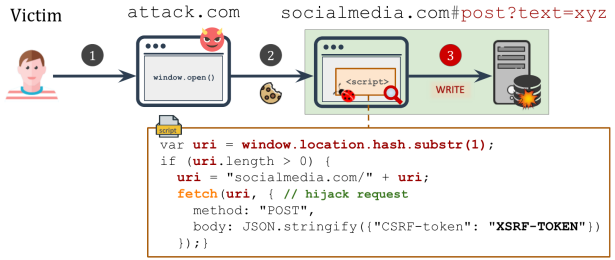
Problem Statement

- Client-side CSRF only one instance of the larger issue of request hijacking
  - Studied client-side CSRF before [USEC'21]
  - Focused
- Other types of HTTP requests and APIs exists
  - The `sendBacon` API accounting for > 35% of the API calls for async reque
  - Web sockets, SSE connections, push notifications, etc
- Attack surfa
  - No web measurement available, in-the-wild prevalence of request hijacking unknown
- Defenses
  - Classical re
  - What countermeasures are useful?

RQ1: Browser APIs and Attacks

RQ2: Detection and Prevalence

RQ3: Defenses and Effectiveness



RQ1: Request Browser APIs

Compile a list of request-sending Web APIs and their capabilities (W3C, WHATWG)

- Configurable fields (e.g., URL, body, headers)
- Network schemes and methods
- Default constraints (e.g., Same-Origin Policy)

Result: identified **10 request APIs** across six broad request types

	API	Req. Type	Specs	Capabilities				
				Schemes	Methods	URL	Body	Header
#1	Location Href	Top-Level Navigation	[38] §7.2.4	HTTP(S), JS	GET	●	○	○
#2	XMLHttpRequest	Async. Request	[39] §3.5	HTTP(S)	Any	●	●	●
#3	sendBeacon	Async. Request	[17] §3.1	HTTP(S)	POST	●	●	○
#4	Window Open	Window Navigation	[38] §7.2.2.1	HTTP(S)	GET	●	○	○
#5	Fetch	Async. Request	[16] §5.4	HTTP(S)	Any	●	●	●
#6	Push	Push Subscription	[40] §3.3	HTTP(S)	GET, POST	●	●	○
#7	WebSocket	Socket Connection	[41] §3.1	WS(S)	GET	●	●	○
#8	Location Assign	Top-Level Navigation	[38] §7.2.4	HTTP(S), JS	GET	●	○	○
#9	Location Replace	Top-Level Navigation	[38] §7.2.4	HTTP(S), JS	GET	●	○	○
#10	EventSource	Server-Sent Event	[38] §9.2	HTTP(S)	GET	●	○	○

RQ1: Vulnerabilities and Attacks

Examined the security impact when an attacker controls one or more API inputs

- Forge asynchronous request URL --- > client-side CSRF, information leaks
- Forge Location URL --- > client-side XSS, open redirections
- …

Result: identified 10 distinct client-side request hijacking vulnerabilities

- Seven new vulnerabilities
- Two new variants (i.e., new API and/or exploitation)

See paper for more!

Vulnerability	Reqs.	CSRF	XSS	WS Hijack	SSE Hijack	Inf. Leak	Open Red.	DoS	Related Ref.
Forge. Async Req. URL	#2, 3, 5	●	○	○	○	●	○	○	[10, 12, 44]
Forge. Async Req. Body	#2, 3, 5	●	○	○	○	○	○	○	[1, 2, 12, 44]
Forge. Async Req. Header	#2, 5	●	○	○	○	○	○	○	-
Forge. Push Req. URL	#6	○	○	○	○	●	○	●	-
Forge. Push Req. Body	#6	●	○	○	○	○	○	○	[45–47]
Forge. EventSource URL	#10	○	○	○	●	●	○	○	[48]
Forge. WebSocket URL	#7	○	○	●	○	●	○	○	-
Forge. WebSocket Body	#7	●	○	●	○	○	○	○	[44, 49–52]
Forge. Location URL	#1, 8, 9	●	●	○	○	○	●	○	[30, 53, 54]
Forge. Window Open URL	#4	●	●	○	○	○	●	○	-

Legend: Forge.= Forgeable; SSE= Server-Sent Event; WS= WebSocket; #i= row i in Table 1; ● = Applicable Attack; ○ = Otherwise.

RQ1: Request API Prevalence

- In total, observed 7.9M API calls in Tranco top 10K domains (~1M webpages)

- **Most widespread**
  - Top-level navigation requests via `location.href`
  - Present on more than 8K sites

- **Most frequently used**
  - Asynchronous requests via the `XMLHttpRequest`
  - Almost 3M calls across over 400K pages

	API	# Sites	# Pages	# Calls
#1	Location Href	8,044	214,554	1,096,306
#2	XMLHttpRequest	7,522	407,819	2,884,556
#3	sendBeacon	7,061	291,580	2,824,388
#4	Window Open	6,972	162,153	559,592
#5	Fetch	5,215	105,463	403,701
#6	Push	1,528	23,566	40,567
#7	WebSocket	1,280	33,724	145,713
#8	Location Assign	987	10,092	22,309
#9	Location Replace	731	6,421	14,309
#10	EventSource	453	1,690	5,503

RQ1: Request API Prevalence

- In total, observed 7.9M API calls in Tranco top 10K domains (~1M webpages)

- **Most widespread**
  - Top-level navigation requests via `location.href`
  - Present on more than 8K sites

- **Most frequently used**
  - Asynchronous requests via the `XMLHttpRequest`
  - Almost 3M calls across over 400K pages.

	API	# Sites	# Pages	# Calls
#1	Location Href	8,044	214,554	1,096,306
#2	XMLHttpRequest	7,522	407,819	2,884,556
#3	sendBeacon	7,061	291,580	2,824,388
#4	Window Open	6,972	162,153	559,592
#5	Fetch	5,215	105,463	403,701
#6	Push	1,528	23,566	40,567
#7	WebSocket	1,280	33,724	145,713
#8	Location Assign	987	10,092	22,309
#9	Location Replace	731	6,421	14,309
#10	EventSource	453	1,690	5,503

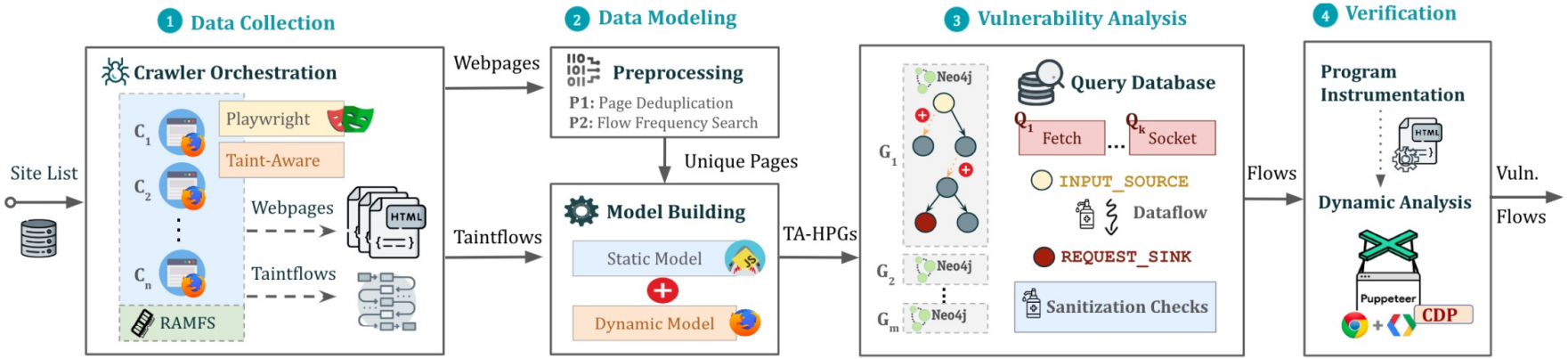
The widespread usage of request-related APIs presents an attractive attack surface

Request hijacking threats have not been considered for 44% of API calls by prior work

RQ2: Vulnerability Detection (JAW v3: Sherriff)

- Proposed a static-dynamic framework to study client-side request hijacking at scale

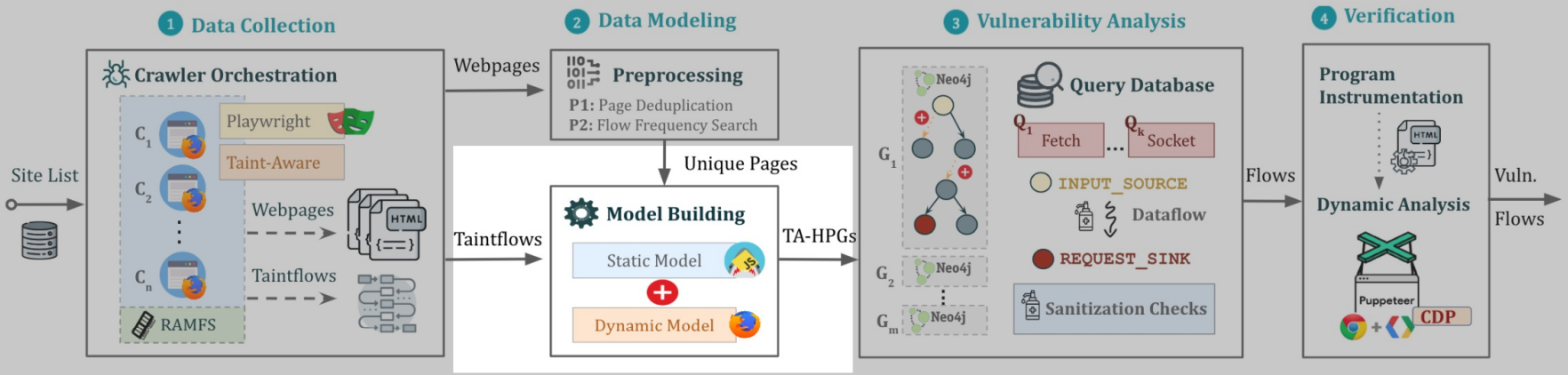
https://ja-w.me



RQ2: Vulnerability Detection (JAW v3: Sherriff)

- Proposed a static-dynamic framework to study client-side request hijacking at scale

https://ja-w.me

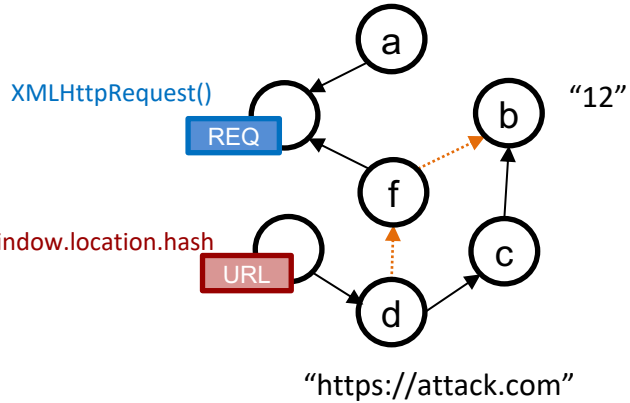


RQ2: Taintflow-Augmented Hybrid Property Graphs

Hybrid Property Graphs

- Static: AST, CFG, PDG, IPCG, ERDDG, …
- Dynamic: Concrete Program Values

Example HPG



Data Flow Analysis

- Track the propagation of **attacker-controlled** values
- Problem: **missing edges** due to static analysis

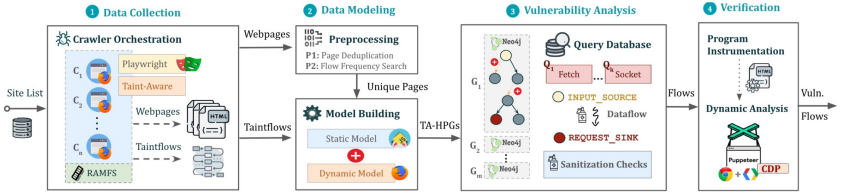
Taintflow-Augmented HPGs

- Use in-browser dynamic taint tracking to **reconstruct missing edges** in HPGs
- Patched Foxhound[1] to support various sinks (e.g., push API, WebSocket, EventSource, etc)

Code: [1] https://github.com/SAP/project-foxhound

RQ2: Vulnerability Prevalence

- Empirical study to quantify the prevalence of client-side request-hijacking in the wild



Testbed

- Tranco top 10K websites, 339.2K unique webpages, 11.5M scripts, 32.4B LoC

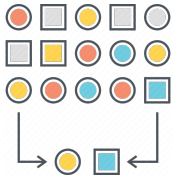
Results

- Detected 202K verified data flows across 17.8K affected pages and 961 sites

The **new vulnerability types and variants** constitute over **36%** of the cases

Dynamic information crucial for detecting **~67%** of the data flows

RQ2: Exploitations



Demonstrate exploitability by focusing on a random subset of data flows

- Two pages from each of the 961 vulnerable sites

Forgeability verification and use in attacks

- **Cross-Site Scripting:** validation of `javascript:` URIs in top-level requests
- **Request Forgery**: inspect server endpoints triggering state changes
- **Information Leak:** request body exposes sensitive data (PIIs, auth keys, and CSRF tokens)
- **Open Redirect:** susceptibility of top-level requests to arbitrary redirections
- …

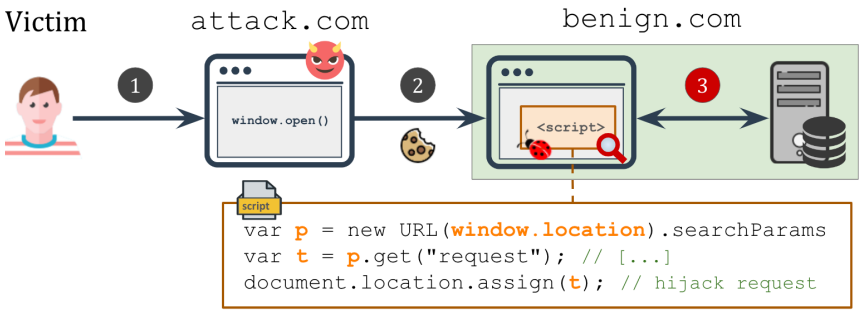
Created PoC exploits for 49 sites

- Microsoft Azure, Starz, Google DoubleClick, VK, DW, and TP-Link
- Arbitrary code execution, account takeover, data exfiltration, open redirections, etc

RQ2: Microsoft Azure Case Study

- Detected a critical request hijacking vulnerability in Microsoft Azure
  - Confirmed and patched (MSRC-79059 VULN-097970)
  - **Impact:** change user settings (**CSRF**), escalated to **client-side XSS**

```
1  var params = (new URL(window.location)).searchParams;
2  var t = params.get("request");
3  if(t != null && t.length){
4      // post message to opener
5      opener && opener.postMessage("reauthPopupOpened", t);
6      // listen for signal
7      window.onmessage = function(){
8          if (event.origin !== opener.origin) return;
9          if (event.data === "sendRequest"){
10             // top-level navigation request
11             document.location.assign(t);}
12     }}
```



RQ2: TP-Link Case Study

- Request hijacking vulnerability in TP-Link escalated to **client-side XSS**
  - Confirmed and patched (TKID240238113)
  - The program performed **no input validation**

TP-Link: page preview functionality

```
1  let $url = new URLSearchParams(location.search)
       .get('url');
2  let $params = location.hash.slice(1).
       toLowerCase();
3  let $product = params.match('&pview=true');
4  if($product && screen.width<=1024){
5      // $url: javascript:alert(1);
6      location.href=$url;}
```

1 Read query param `url`

2 Write `url` to `location.href`

Defenses and their Effectiveness (1 / 3)

Policy-based

Content Security Policy

`connect-src` directive:

- (+) constrains request endpoints to **trusted domains** (i.e., no data exfiltration)
- (-) does not prevent request hijacks for CSRF attacks (i.e., **same-site** endpoints)

Even with a correct configuration:

~**41%** of vulnerabilities **cannot be mitigated** by CSP

Defenses and their Effectiveness (2 / 3)

Policy-based

Content Security Policy

Cross-Origin Opener Policy

`connect-src` directive:

- (+) constrains request endpoints to trusted domains (i.e., no data exfiltration)
- (-) does not prevent request hijacks for CSRF attacks (i.e., same-site endpoints)

Even with a correct configuration:

~41% of vulnerabilities cannot be mitigated by CSP

COOP: window.open() API

- (+) restricts the browsing context to same-origin documents
- (-) **only effective** when window.open() is used for providing malicious input

~**93%** of detected vulnerabilities cannot be mitigated by COOP

Defenses and their Effectiveness (2 / 3)

Policy-based

Content Security Policy

Cross-Origin Opener Policy

Cross-Origin Embedder Policy

Fetch MetaData

See paper for more

`connect-src` directive:

- (+) constrains request endpoints to **trusted domains** (i.e., no data exfiltration)
- (-) does not prevent request hijacks for CSRF attacks (i.e., **same-site** endpoints)

Even with a correct configuration:

⚠ ~**41%** of vulnerabilities **cannot be mitigated** by CSP

COOP: window.open() API

- (+) restricts the browsing context to same-origin documents
- (-) **only effective** when window.open() is used for providing malicious input

⚠ ~**93%** of detected vulnerabilities cannot be mitigated by COOP

Defenses and their Effectiveness (3 / 3)

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Policy-based

Content Security Policy
Cross-Origin Opener Policy
Cross-Origin Embedder Policy
Fetch MetaData

Custom

Input validation

Analyzed vulnerable flows to detect insecure input validation patterns

- Eight distinct behaviours across **three** types of issues

1

Missing checks: ~**47%** of vulnerable data flows

2

Insufficient:

- Trivial checks, e.g., length, type, not null, etc (**~13%**)
- Substring searches and check of URL fields (**~24%**)

```
s.indexOf("benign.com") -> benign.com.evil.com
```

3

Flawed:

- Compare two attacker-controlled values with one another (**~3%**) :

```
QueryParam === window.name
```

Conclusion

Thank You!

- Client-side CSRF is only the **tip of the iceberg**
- Request hijacking data flows are **ubiquitous** (i.e., **9.6%** of sites)
- Request hijacking can have **diverse consequences**
- Existing defenses necessary but **insufficient**

@Soheil__K

https://ja-w.me

https://github.com/SAP/project-foxhound