SCAN ME

JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals

Soheil Khodayari
soheil.khodayari@cispa.saarland

CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

February 24, 2021

About Me

Soheil Khodayari

2nd Year PhD Student @CISPA, Germany (2019 – Present)

Research Group of Dr. Giancarlo Pellegrino

Web Security, Program Analysis

Double MSc. in Computer Science (2017-2019)

- Polytechnic University of Madrid - Technical University of Kaiserslautern
- Previously, researcher @IMDEA
- Supervisor: Prof. Juan Caballero

Publications in NDSS, USENIX Security

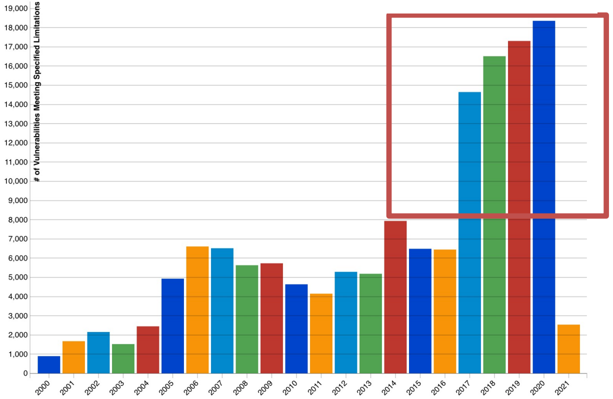
Web Applications

- We know that webapp vulnerability detection is critical
  - Over 4.8 billion websites online, 1.8 billion users [1]
  - Contain a variety of security-sensitive data

Banking Shopping Education

- The complexity of webapps are rising.
  - **Problem:** Existing vulnerability detection tools fall short of capturing this complexity.

Webapp CVEs By Year [2]

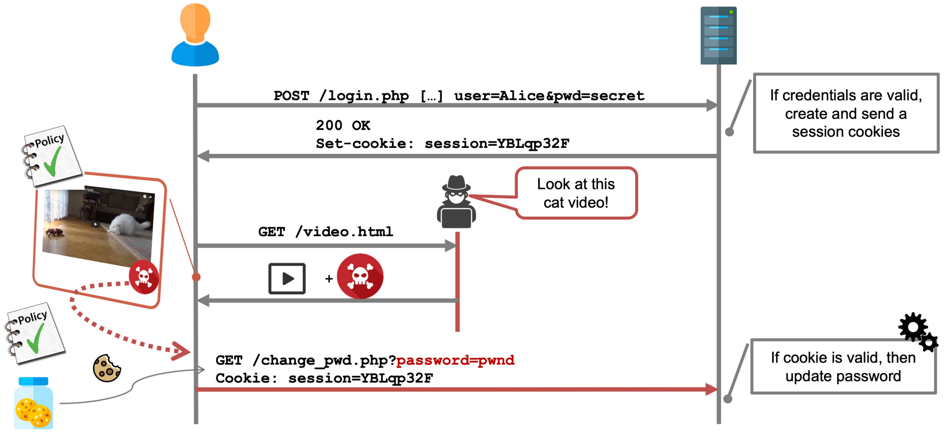


Sources:
1 internetlivestats.com
2 nvd.nist.gov

Cross-Site Request Forgery (CSRF)

- CSRF is an instance of the confused deputy problem
  - Attackers can trick the browser to send a forged request to a target site without the victim intention.

- **Defenses**
  - Referrer/Origin Checks
  - Hard-to-guess parameter
    - Synchronizer tokens
    - HMAC-based tokens
    - Double submit cookie
  - Custom HTTP Headers
  - Same-Site Cookies
    - *SameSite=Lax* cookies by default

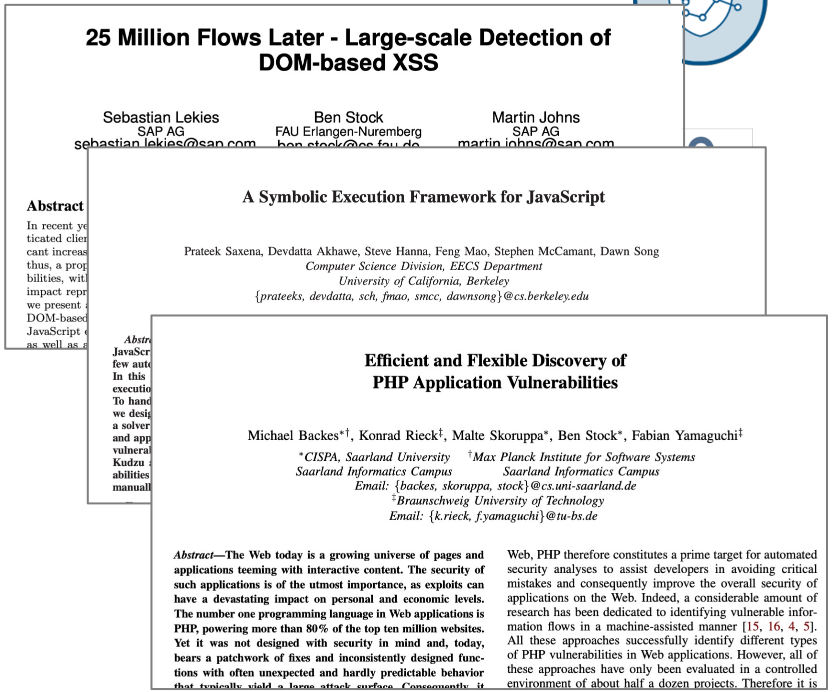


Client-side CSRF: Existing Defenses Are Ineffective!

- Attacker tricks the client-side JS to send a forged request to a target site by manipulating the program's input parameters.

Challenges: Security Analysis of Webapps

- **(C1)** Vulnerability-specific analysis tools and techniques
- **(C2)** Isolated client/server-side security analysis
- **(C3)** Language-specific analysis tools
  - No static canonical representational model for all
  - Event-driven programming languages
- **(C4)** Web execution environment
- **(C5)** Modeling shared third-party code



25 Million Flows Later - Large-scale Detection of DOM-based XSS

Sebastian Lekies
SAP AG

Ben Stock
FAU Erlangen-Nuremberg

Martin Johns
SAP AG

A Symbolic Execution Framework for JavaScript

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, Dawn Song
Computer Science Division, EECS Department
University of California, Berkeley
{prateeks, devdatta, sch, fmao, smcc, dawnsong}@cs.berkeley.edu

Efficient and Flexible Discovery of PHP Application Vulnerabilities

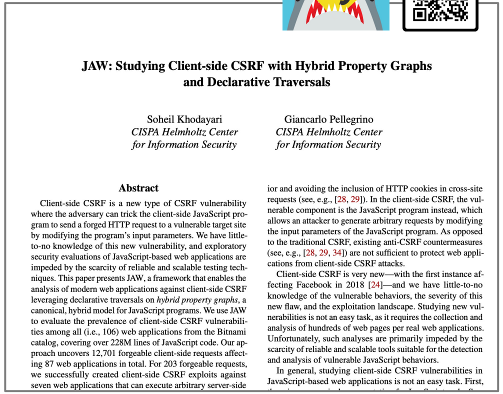
Michael Backes[*†], Konrad Rieck[‡], Malte Skoruppa[*], Ben Stock[*], Fabian Yamaguchi[‡]
[*]CISPA, Saarland University *[†]Max Planck Institute for Software Systems*
Saarland Informatics Campus *Saarland Informatics Campus*
Email: {backes, skoruppa, stock}@cs.uni-saarland.de
[‡]Braunschweig University of Technology
Email: {k.rieck, f.yamaguchi}@tu-bs.de

***Abstract*—The Web today is a growing universe of pages and applications teeming with interactive content. The security of such applications is of the utmost importance, as exploits can have a devastating impact on personal and economic levels. The number one programming language in Web applications is PHP, powering more than 80% of the top ten million websites. Yet it was not designed with security in mind and, today, bears a patchwork of fixes and inconsistently designed functions with often unexpected and hardly predictable behavior**

Web, PHP therefore constitutes a prime target for automated security analyses to assist developers in avoiding critical mistakes and consequently improve the overall security of applications on the Web. Indeed, a considerable amount of research has been dedicated to identifying vulnerable information flows in a machine-assisted manner [15, 16, 4, 5]. All these approaches successfully identify different types of PHP vulnerabilities in Web applications. However, all of these approaches have only been evaluated in a controlled environment of about half a dozen projects. Therefore it is

Contributions: Revisiting the Challenges

- **(C1)** Vulnerability-specific analysis tools and techniques
  - We decouple the code representation from analysis.
  - Focus on client-side CSRF, generalization to other on the way, e.g., XSS, DOM clobbering, etc.
- **(C2)** Isolated client/server-side security analysis
- **(C3)** Language-specific analysis tools
  - Hybrid Property Graphs (HPGs), canonical representation for JS + Event-Driven paradigm
  - Support for other languages on the way, e.g., Python, PHP, etc.
- **(C4)** Web execution environment
  - HPGs capture the dynamics of the execution env via snapshots of the web env (e.g., DOM trees) and traces of JS events
- **(C5)** Modeling shared third-party code
  - We generate reusable symbolic models of external libraries.



JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals

Soheil Khodayari
CISPA Helmholtz Center for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security

Abstract

Client-side CSRF is a new type of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by modifying the program's input parameters. We have little-to-no knowledge of this new vulnerability, and exploratory security evaluations of JavaScript-based web applications are impeded by the scarcity of reliable and scalable testing techniques. This paper presents JAW, a framework that enables the analysis of modern web applications against client-side CSRF leveraging declarative traversals on *hybrid property graphs*, a canonical, hybrid model for JavaScript programs. We use JAW to evaluate the prevalence of client-side CSRF vulnerabilities among all (i.e., 106) web applications from the Bitnami catalog, covering over 228M lines of JavaScript code. Our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications in total. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side

ior and avoiding the inclusion of HTTP cookies in cross-site requests (see, e.g., [28, 29]). In the client-side CSRF, the vulnerable component is the JavaScript program instead, which allows an attacker to generate arbitrary requests by modifying the input parameters of the JavaScript program. As opposed to the traditional CSRF, existing anti-CSRF countermeasures (see, e.g., [28, 29, 34]) are not sufficient to protect web applications from client-side CSRF attacks.

Client-side CSRF is very new—with the first instance affecting Facebook in 2018 [24]—and we have little-to-no knowledge of the vulnerable behaviors, the severity of this new flaw, and the exploitation landscape. Studying new vulnerabilities is not an easy task, as it requires the collection and analysis of hundreds of web pages per real web applications. Unfortunately, such analyses are primarily impeded by the scarcity of reliable and scalable tools suitable for the detection and analysis of vulnerable JavaScript behaviors.

In general, studying client-side CSRF vulnerabilities in JavaScript-based web applications is not an easy task. First,

SCAN ME

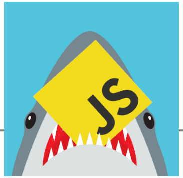
To be appeared in USENIX Security'21

Contributions

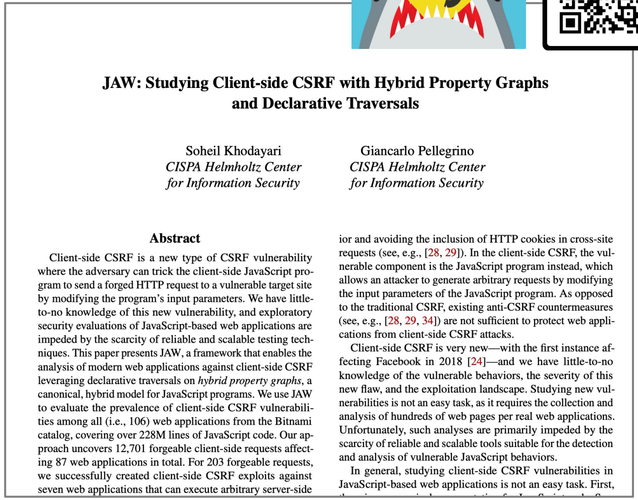
- Presented JAW, a framework that detects client-side CSRF by instantiating a HPG for each web page.

- Evaluated JAW with 228M LoC of 106 popular applications from the Bitnami catalog.

- First systematic study of client-side CSRF and taxonomy of forgeable client-side requests.
  - Identified 12,701 forgeable requests affecting 87 applications.



SCAN ME



JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals

Soheil Khodayari
CISPA Helmholtz Center for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security

Abstract

Client-side CSRF is a new type of CSRF vulnerability where the adversary can trick the client-side JavaScript program to send a forged HTTP request to a vulnerable target site by modifying the program's input parameters. We have little-to-no knowledge of this new vulnerability, and exploratory security evaluations of JavaScript-based web applications are impeded by the scarcity of reliable and scalable testing techniques. This paper presents JAW, a framework that enables the analysis of modern web applications against client-side CSRF leveraging declarative traversals on *hybrid property graphs*, a canonical, hybrid model for JavaScript programs. We use JAW to evaluate the prevalence of client-side CSRF vulnerabilities among all (i.e., 106) web applications from the Bitnami catalog, covering over 228M lines of JavaScript code. Our approach uncovers 12,701 forgeable client-side requests affecting 87 web applications in total. For 203 forgeable requests, we successfully created client-side CSRF exploits against seven web applications that can execute arbitrary server-side

ior and avoiding the inclusion of HTTP cookies in cross-site requests (see, e.g., [28, 29]). In the client-side CSRF, the vulnerable component is the JavaScript program instead, which allows an attacker to generate arbitrary requests by modifying the input parameters of the JavaScript program. As opposed to the traditional CSRF, existing anti-CSRF countermeasures (see, e.g., [28, 29, 34]) are not sufficient to protect web applications from client-side CSRF attacks.

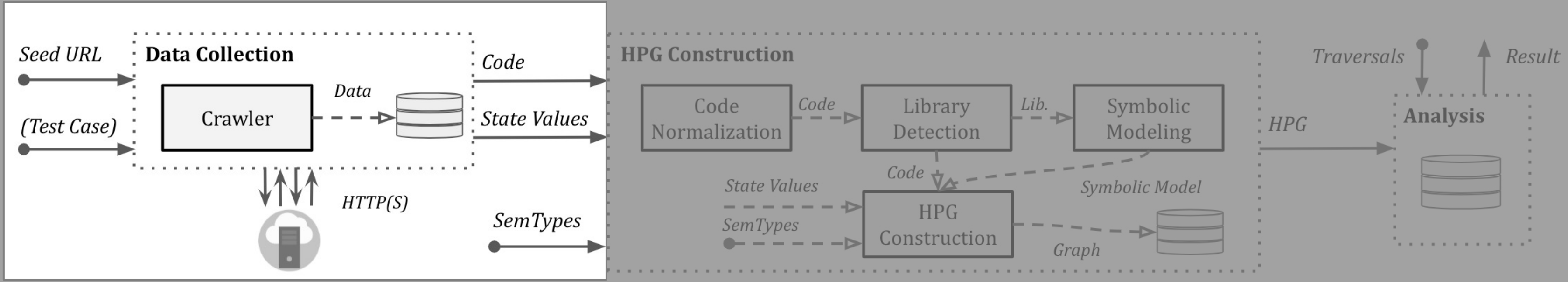
Client-side CSRF is very new—with the first instance affecting Facebook in 2018 [24]—and we have little-to-no knowledge of the vulnerable behaviors, the severity of this new flaw, and the exploitation landscape. Studying new vulnerabilities is not an easy task, as it requires the collection and analysis of hundreds of web pages per real web applications. Unfortunately, such analyses are primarily impeded by the scarcity of reliable and scalable tools suitable for the detection and analysis of vulnerable JavaScript behaviors.

In general, studying client-side CSRF vulnerabilities in JavaScript-based web applications is not an easy task. First,

https://soheilkhodayari.github.io/JAW

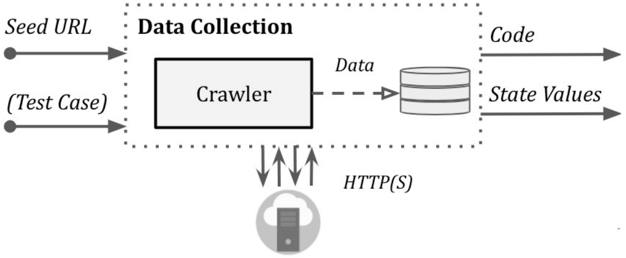
JAW: Approach Overview

A. Data Collection
B. Graph Construction
C. Analysis Traversals



JAW: Data Collection

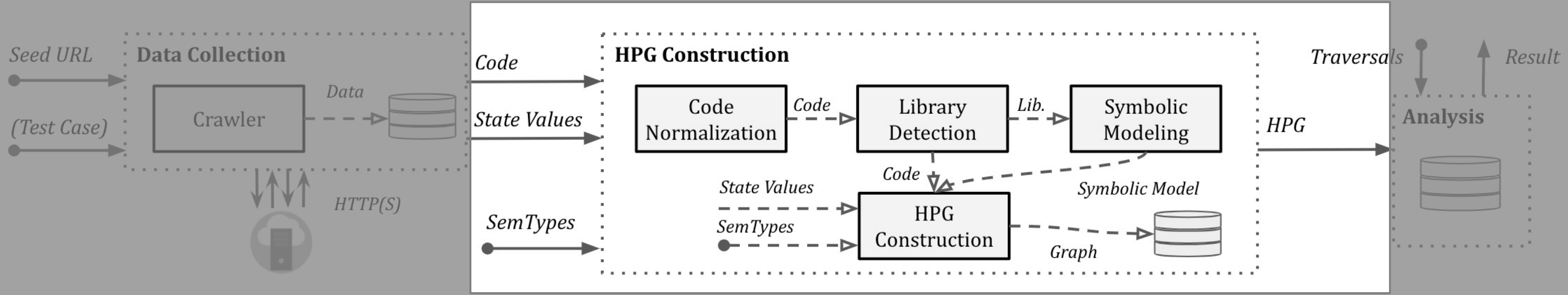
- Chrome-based crawler with Selenium
- Enhanced with chrome extensions
- **Outputs:**
  - JavaScript Code
  - HTTP Requests and Responses
  - Dynamically Fired Events
  - Concrete snapshots of the global *Window* object
    - window.document (DOM tree)
    - window.localStorage
    - window.document.cookie
    - …



Dynamic Information

JAW: Approach Overview

A. Data Collection
B. Graph Construction
C. Analysis Traversals



Hybrid Property Graphs (HPGs): Building Blocks

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

- **Code Representation**
  - Abstract Syntax Tree (AST)
  - Control Flow Graph (CFG)
  - Program Dependence Graph (PDG)
  - Inter-Procedural Call Graph (IPCG)
  - Event Registration, Dispatch and Dependency Graph (ERDDG)
  - Semantic Types and Symbolic Models

CPG for C/C++
[Yamaguchi, S&P'14]

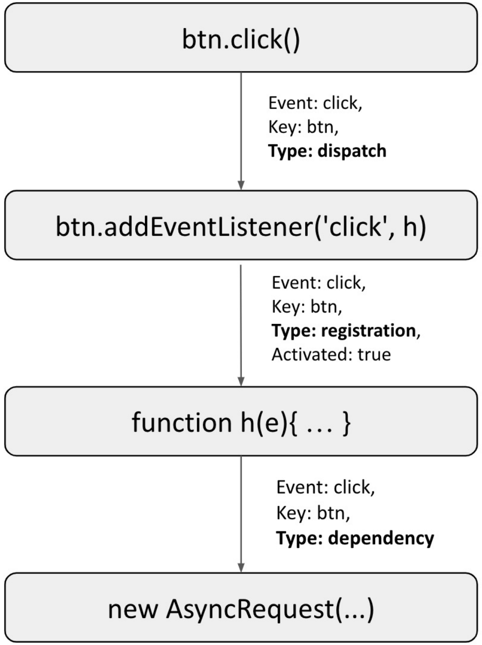
CPG for PHP
[Bakes et al., EuroS&P'17]

- **State Values**
  - Event Traces
  - Environment Properties

Event Registration, Dispatch and Dependency Graph

- **Problem:** when an event is dispatched, one or more registered functions are executed
  - Can change the state of the program
  - Register new handlers
  - Fire new events
- **Solution:** the ERDDG

```
var btn = document.querySelector("button");
function h(e){
        new AsyncRequest(...);
}
btn.addEventListener("click", h);
// [...]
btn.click();
```



Symbolic Models

- External libraries: over 60% of the total LoC of each webpage.
- **Problem:**
  - Existing approaches: Inefficient, include library code in the analysis
- **Goal:** Shared library code can be modeled once and re-used.
  - Extract a symbolic model from each library and use it as a proxy.
  - The symbolic model is an assignment of a label to library constructs. → **Semantic Types**
- Example:
  - **"REQ"** for all functions that send HTTP requests, e.g., "*asyncRequest*" of YUI library
  - **"WIN.LOC"** for library functions consuming "*window.location"*
  - **"WEB-STORAGE"** for library functions consuming "localStorage/sessionStorage"

Symbolic Models (Cont'd)

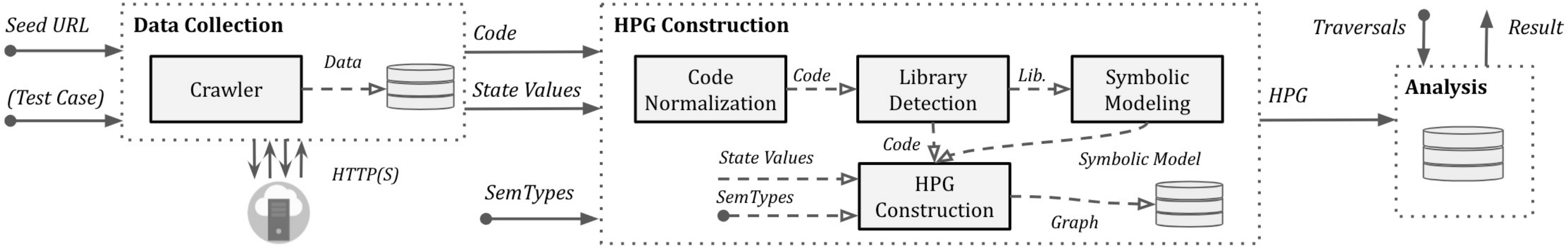
- To reconstruct the data flow of programs that use library functions, we define two semantic types:

    - Type "o < --- i": function(i){ return o = g(i); }

    - Type "o ~ i" function(i){ if(cond(i)) return o; }

JAW: Approach Overview

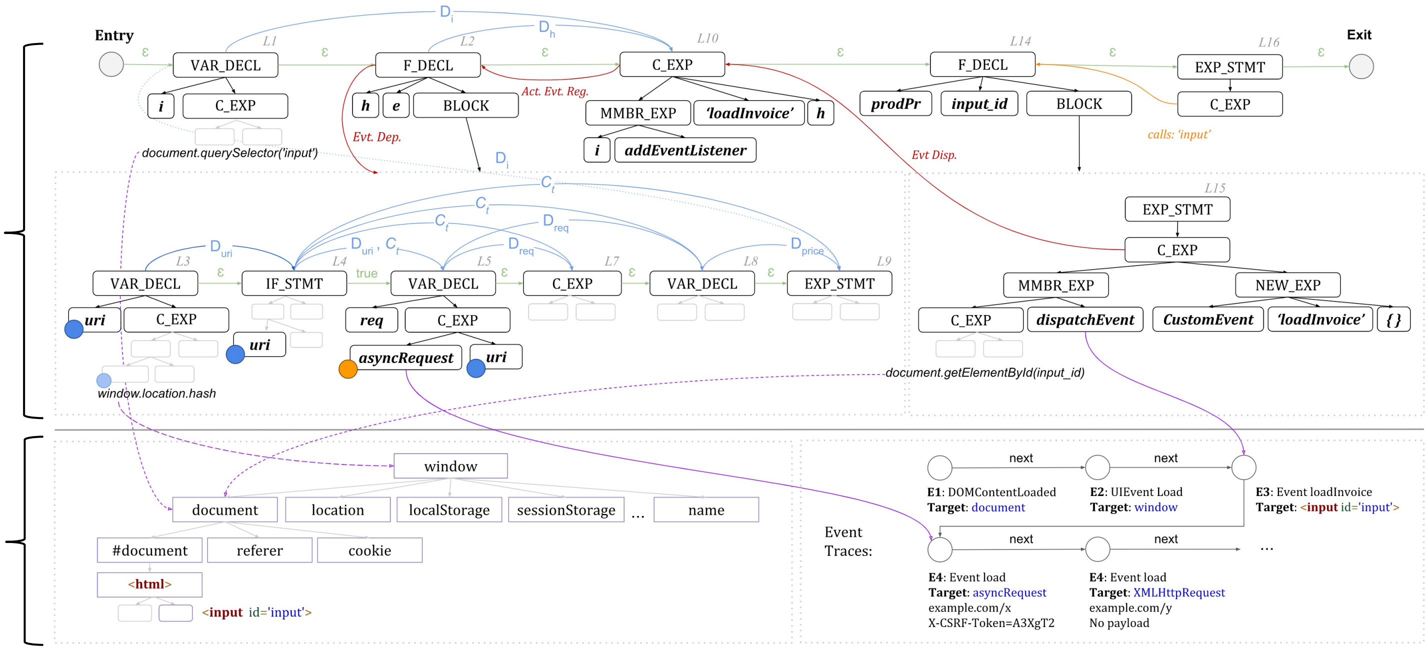
A. Data Collection
B. Graph Construction
C. Analysis Traversals



Example: Hybrid Property Graphs

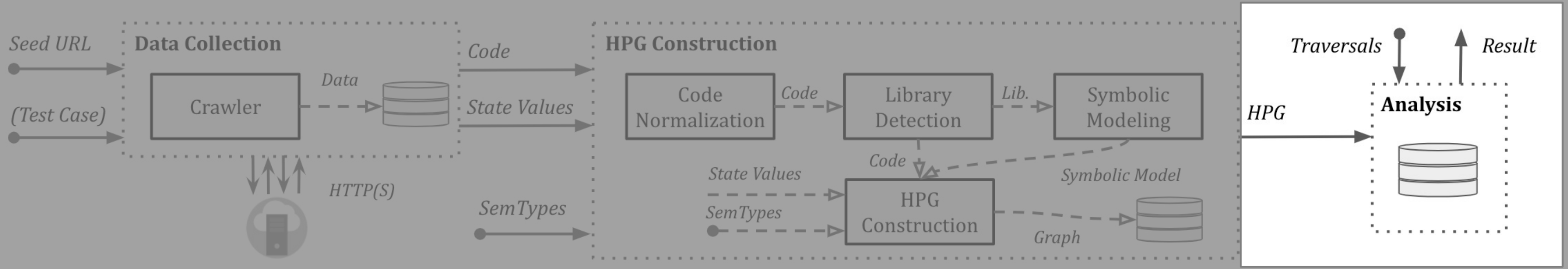
Code Repr.
(Static Part)

State Values
(Dynamic Part)



JAW: Approach Overview

A. Data Collection
B. Graph Construction
C. Analysis Traversals



Analysis: Vulnerability Detection

- **Client-side CSRF**
  A. Data flow from an attacker-controlled input to a parameter of a request <u>R</u>.
    - lines of code having both **“WIN.LOC”** and **“REQ”** semantic types.

  B. <u>R</u> is reachable at page load.

1
```
var domain = validate_domain(window.location.hash);
fetch("https://" + domain + "/"); <-
```

2
```
var uri = window.location.hash;
var xhr = new XMLHttpRequest();
xhr.open("GET", uri); <-
xhr.send();
```

3
```
var uri = validate_uri(window.location.hash);
var xhr = new extLibraryHttpRequest();
let a = xhr; // alias
a.open("POST", uri); <-
var q = window.location.search;
a.send("q1="+q.substr(1,10)+ "&q2="+q.substr(11); <-
```

- Model both conditions using declarative traversals
  - A query Q contains all nodes n of HPG for which a predicate P is true: $Q = \{n : P(n)\}$

$$
\begin{aligned}
Q_A = \{ n :\ & \mathrm{isDeclOrStmt}(n) \land \exists c1, c2, c1 \mathrel{!=} c2 \\
& \land \mathrm{hasChild}(n, c1) \land \mathrm{hasSemType}(c1, \text{“REQ”}), \\
& \land \mathrm{hasChild}(n, c2) \land \mathrm{hasSemType}(c2, \text{“WIN.LOC”}) \\
\}
\end{aligned}
$$

Evaluation: Experimental Setup

- Tested all webapps (i.e., 106) from the Bitnami catalog
  - Ready-to-deploy containers of preconfigured web applications.
  - Why Bitnami?
    - Popularity
    - Diversity
    - Use by prior work [Pellegrino et al., CCS’17]
  - For each webapp, we created:
    - One user account for each supported levels of privilege.
    - A Selenium state script to perform the login.
      - A total of 136 scripts, 1-5 per webapp
  - Instantiated JAW against each webapp by inputing a single seed URL.

Evaluation: Forgeable Requests

- A total of 12,701 forgeable requests
- **Exploitations:**
  - Manually looked for practical exploitations in 516 requests:
    - Selected all requests across all groups, except for “DOM.READ” type.
    - for “DOM.READ”, we focused on one randomly selected request per webapp.

Sources	**Forgeable**	**Apps**
DOM.COOKIES	67	5
DOM.READ	12,268	83
*-STORAGE	76	8
DOC.REFERRER	1	1
POST-MESSAGE	8	8
WIN.NAME	1	1
WIN.LOC	280	12
Total forgeable	12,701	87
Non-reachable code	36,665	101
Total	49,366	106

  - Created a working exploit for 203 forgeable requests affecting seven web applications:
    - SuiteCRM, SugarCRM, Neos, Kibana, Modx, Odoo, Shopware
    - Account takeover, deleting user assets, executing malicious queries, etc.
    - All exploits use data values of WIN.LOC, that can be forged by any web attacker.

Evaluation: Analysis of Forgeable Requests

- Exploitation landscape can be influenced by:
  - Degree of attacker's control on forgeable requests
- In total, identified 25 distinct templates
  - The majority of webapps use only one (i.e., 68 apps) or two (i.e., 17 apps) templates across all their webpages
- Request Fields:
  - In total, 55, 34, and 12 webapps allow modifying one, more than one, and all fields, respectively.

Outgoing HTTP Request						Total	
Dom.	**Path**	**Query**	**Body**	**Part**	**Control**	**Reqs**	**Apps**
		✓		One	-, A, -	16	11
			✓	One	-, A, -	5	5
			✓	One	W, -, -	(*)166	25
			✓	One	-, -, P	1	1
	✓			One	W, -, -	28	1
	✓			One	-, A, -	7	7
	✓			One	-, -, P	6	6
		✓		One	-, -, P	11	11
	✓		✓	Mult	-, A, -	4	1
	✓		✓	Mult	W, -, -	(*)20	1
	✓	✓		Mult	W, A, P	6	1
		✓	✓	Mult	W, -, -	2	1
		✓		Mult	-, A, -	7	7
			✓	Mult	-, -, P	2	2
	✓			Mult	-, A, -	3	3
		✓		Mult	-, -, P	1	1
			✓	Mult	-, A, -	5	5
	✓			Mult	-, -, P	6	6
			✓	Mult	W, -, -	28	8
	✓	✓		Any	W, -, -	1	1
✓	✓	✓		Any	W, -, -	(*)185	8
✓	✓	✓	✓	Any	W, -, -	1	1
			✓	Any	W, -, -	(*)1	1
			✓	Any	W, -, -	2	2
	✓	✓	✓	Any	W, -, -	1	1

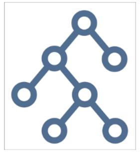
Legend: A=Appending; P=Prepending; W=Writing.

JAW Is Only the First Step. What’s Next?

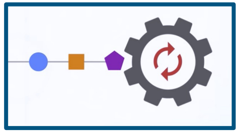
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

- **(C1)** Vulnerability-specific analysis tools and techniques
  - Support for additional vulnerability classes on the way.

- **(C2)** Isolated client/server-side security analysis
  - Web Property Graphs (WPGs)
    - Connecting the client-side to the server-side program in the property graph.



- **(C3)** Language-specific analysis tools
  - Support for other programming languages on the way.
  - Language-agnostic property graphs, requires UAST.



- **(C4)** Web execution environment
- **(C5)** Modeling shared code
  - Incremental Static Analysis

Conclusion

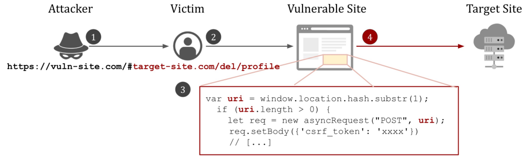
https://soheilkhodayari.github.io/JAW

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Thank You!

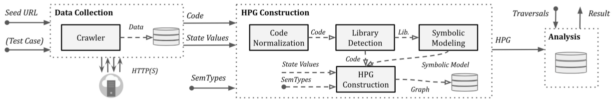
Client-side CSRF: Existing Defenses Are Ineffective!

- Attacker tricks the client-side JS to send a forged request to a target site by manipulating the program's input parameters.

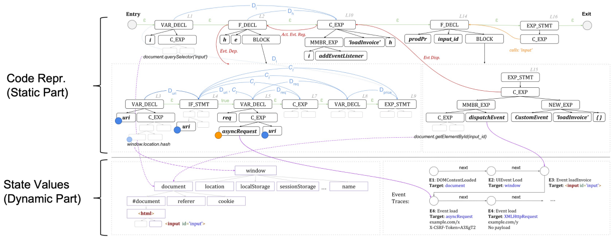


JAW: Approach Overview

A. Data Collection
B. Graph Construction
C. Analysis Traversals



Example: Hybrid Property Graphs



Evaluation: Forgeable Requests

- A total of 12,701 forgeable requests
- **Exploitations:**
  - Manually looked for practical exploitations in 516 requests:
    - Selected all requests across all groups, except for "DOM.READ" type.
    - for "DOM.READ", we focused on one randomly selected request per webapp.
  - Created a working exploit for 203 forgeable requests affecting seven web applications:
    - SuiteCRM, SugarCRM, Neos, Kibana, Modx, Odoo, Shopware
    - Account takeover, deleting user assets, executing malicious queries, etc.
    - All exploits use data values of WIN.LOC, that can be forged by any web attacker.

Sources	Forgeable	Apps
DOM.COOKIES	67	5
DOM.READ	12,268	83
*-STORAGE	76	8
DOC.REFERRER	1	1
POST-MESSAGE	8	8
WIN.NAME	1	1
WIN.LOC	280	12
Total forgeable	12,701	87
Non-reachable code	36,665	101
Total	49,366	106

EXPLOIT

SCAN ME