PMForce

Systematically Analyzing postMessage Handlers at Scale

Steffens, M., & Stock, B.

CS 568, Fall 2020

Presenter: Ashesh Singh

1. Background

1.1 Background: Same Origin Policy (1/2)

- The **same-origin policy (SOP)** restricts how a document or script loaded from one *origin* can interact with a resource from another *origin*.

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Same origin	Only the path differs
`http://store.company.com/dir/inner/another.html`	Same origin	Only the path differs
`https://store.company.com/page.html`	Failure	Different protocol
`http://store.company.com:81/dir/page.html`	Failure	Different port (`http://` is port 80 by default)
`http://news.company.com/dir/page.html`	Failure	Different host

Comparison with http://store.company.com/dir/page.html

1.1 Background: Same Origin Policy (2/2)

- We have seen that bypassing SOP (intentionally or unintentionally) can case security issues, eg:

  Watanabe, T., Shioji, E., Akiyama, M., & Mori, T. *Melting Pot of Origins: Compromising the Intermediary Web Services that Rehost Websites.*

1.2 Javascript

- JS is a dynamic, weakly typed language with implicit type expressions.
- Type safety is verified at runtime (dynamic).

https://repl.it/@asing80/UnequaledWavyQueryoptimizer

https://repl.it/@asing80/BadTrickyControlpanel

1.3 Symbolic Execution

- Generate a set of input values that would lead to program execution.
- Framework/tools: ExpoSEJS/ExpoSE

```
var value1 = document.getElementById("value1").value // "some-valid-value1"
var value2 = document.getElementById("value2").value // "some-valid-value2"

if (value1 === "some-valid-value1" || value2 === "some-valid-value2") {
 console.log("Hello!")
}
```

1.4 Forced Execution

- Modify the program (or control flow) that would lead to program execution.

```
var value1 = document.getElementById("value1").value // "some-valid-value1"
var value2 = document.getElementById("value2").value // "some-valid-value2"

if (value1 === "some-valid-value1" || value2 === "some-valid-value2") {
 console.log("Hello!")
}
```

1.5 Taint Analysis

- Checks/marks computations that are affected by predefined taint sources such as user input.

2. Motivation

2.1 Motivation: What is this paper about?

- Finding vulnerable `postMessage` handlers (how *many*?)
- Wide-scale study using an automated framework

  https://github.com/mariussteffens/pmforce

2.1.1 Motivation: What is `postMessage`?

- A method on the Windows Web API that enables “safe” cross-origin communication between Window objects.

  ```
  window.postMessage(...)
  // where `window` is a reference to the current Window Object
  ```

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2														
6-7		[3] 3-5														
[1] 8-9		[4] 6-7		3.1-3.2												
[2] 10	12-85	[4 5] 8-81	4-85	4-13.1	10-71	3.2-13.7		2.1-4.4.4	12-12.1				4-11.2			
[2] 11	86	[4 5] 82	86	14	72	14	all	81	59	86	[4 5] 82	12.12	12.0	10.4	7.12	2.5
		[4 5] 83-84	87-89	TP												

Browser support for Window API: postMessage (https://caniuse.com/mdn-api_window_postmessage)

2.1.2 Motivation: What is `postMessage` Handler?

- The “handler” is the part of code (usually on a different origin, on separate tab/iframe/window) that interprets(?) this post message.
- Optionally, it will respond back with another `postMessage`.

The API method by itself is not unsafe.

2.1.2 Motivation: Need for `postMessage`?

- Convenience

  Examples:

  https://pminitiator1.netlify.app/

  https://asing80.people.uic.edu/cs568/presentation/demo/postmessage-handler-strict.html

2.2 Motivation: Why bother investigating?

- Increased usage
- Intrinsic vulnerability (specially in the handler)

3. Attack Models

3.1 Attack Models: Cross Site Scripting (XSS)

- A type of injection attack
- Attacker is able to (inject and) execute malicious code on a benign website.
- String-to-code conversion:
  - `eval()`
  - `Document.write()`
  - `Element.innerHTML`

3.2 Attack Models: State Manipulation

- Remember, HTTP protocol is stateless.
- Attacker is able to manipulate `cookies` or `localStorage`.
- Can even circumvent CSRF protection.

3.3 Attack Models: PM Origin Laundering

- In a `postMessage` (PM) relay setup, attacker can target an otherwise secure handler by going through an insure one.

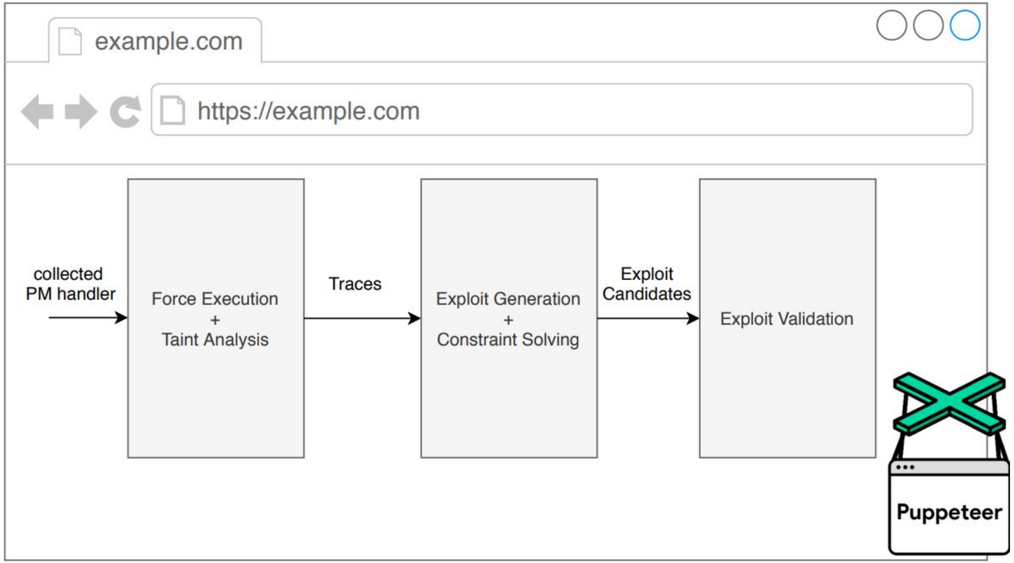
3.4 Attack Models: Privacy Leaks

- If the handler sends back acknowledgement (to the source origin) as another `postMessage`, they risk revealing private information.

4. Methodology

4.1 Methodology: Overview

`postMessage` handlers of the top 100,000 sites, according to Tranco.



4.2 Methodology: Iroh (forced execution)   (1/2)

- Iroh, dynamic code analysis tool for JavaScript.

  Examples: https://maierfelix.github.io/Iroh/examples/index.html

4.2 Methodology: Iroh (forced execution) (2/2)

- Authors take care of only focusing on the handlers and minimize issues due to side effects.

4.3 Methodology: Taint Analysis (1/2)

- Checks/marks computations that are affected by predefined taint sources such as user input.
- Authors create `Proxy objects` as input to capture operations performed on them.

4.3 Methodology: Taint Analysis (2/2)


```
{
        "ops": [
          {
            "type": "ops_on_parent_element",
            "old_ops": [],
            "old_identifier": "event"
          },
          {
            "args": [
              0,
              8
            ],
            "type": "member_function",
            "function_name": "substring"
          },
          {
            "op": "===",
            "val": "https://",
            "side": "left",
            "type": "Binary"
          }
        ],
        "identifier": "event.origin"
}
```

4.4 Methodology: Constraint Solving (z3)   (1/2)

- Transform output of taint analysis into clauses.
- Solve using Z3
- Translate back into JavaScript

Example (Z3 python):

```
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

5. Automated Validation

5. Automated Validation

- Solved constraints translated back to Javascript.
- Exploit templates used to target sample code.
- Payload sent to target handlers and logged.

6. Results

Sink	total number of handlers	number of unique handlers	vulnerable handlers		**with** origin check		**without** origin check	
			number	*sites*	number	*sites*	number	*sites*
eval	132	57	43	*166*	18	*110*	25	*56*
insertAdjacentHTML	38	4	4	*12*	1	*1*	3	*11*
innerHTML	37	37	16	*54*	4	*35*	12	*19*
document.write	26	4	3	*5*	2	*4*	1	*1*
scriptTextContent	4	4	1	*3*	0	*0*	1	*3*
jQuery .html	3	3	1	*1*	0	*0*	1	*1*
sum code execution	217	105	66	*240*	24	*149*	43	*91*
set cookie	108	101	18	*110*	2	*4*	16	*106*
localStorage	63	60	30	*31*	7	*8*	23	*23*
sum state manipulation	161	150	47	*140*	9	*12*	38	*128*
total sum	377	252	111	*379*	32	*160*	80	*219*

Compared to a previous 2013 study, 24 of 32 handlers perform strict origin checks.

6. Comments

- A technically involved paper. Difficult to navigate without understanding of concepts like taint analysis, constraint solving.
- Privacy leaks and laundering were discussed but not analyzed.
- Highly dependent on functionality (or lack of, ie. regex handling) provided by Z3.