Parse Me Baby One More Time: Bypassing HTML Sanitizer via Parsing Differentials

David Klein

Institute for Application Security
Technische Universität Braunschweig

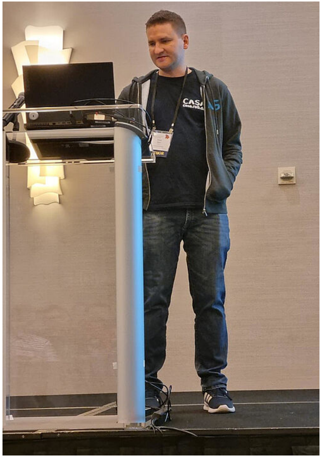
david.klein@tu-braunschweig.de

IAS INSTITUTE FOR APPLICATION SECURITY

CASA CYBER SECURITY IN THE AGE OF LARGE-SCALE ADVERSARIES

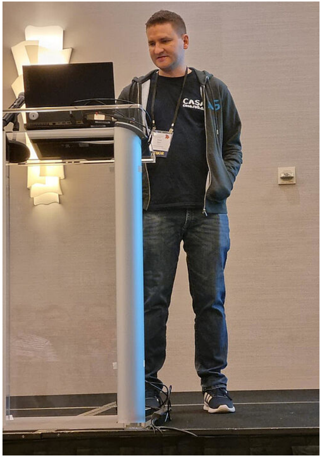
Technische Universität Braunschweig

About Me



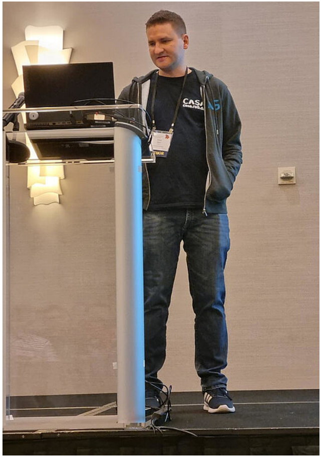
- PhD Candidate
  - At TU Braunschweig
  - Group of Martin Johns

About Me



- PhD Candidate
  - At TU Braunschweig
  - Group of Martin Johns
- Research interests:
  - Web Security
  - Privacy
  - Application Security

About Me



- PhD Candidate
  - At TU Braunschweig
  - Group of Martin Johns
- Research interests:
  - Web Security
  - Privacy
  - Application Security
- Soon on the Academic Job Market

Cross Site Scripting (XSS)

Client-Side

```
document.write(location.hash);
```

Server-Side

```
<?php
echo $_GET["name"];
```

Cross Site Scripting (XSS)

Client-Side

```
document.write(location.hash);
```

User Input

Server-Side

```
<?php
echo $_GET["name"];
```

User Input

Cross Site Scripting (XSS)

Client-Side

```
document.write(location.hash);
```

Reflection

Server-Side

```
<?php
echo $_GET["name"];
```

Reflection

Cross Site Scripting (XSS)

Client-Side

```
document.write(location.hash);
```

Server-Side

```
<?php
echo $_GET["name"];
```

Such Code Patterns Are Everywhere!

Cross Site Scripting (XSS)

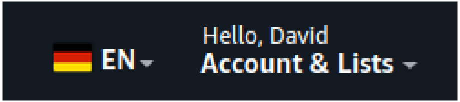
Client-Side

```
document.write(location.hash);
```

Server-Side

```
<?php
echo $_GET["name"];
```

Such Code Patterns Are Everywhere!



EN

Hello, David
Account & Lists

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Server-Side

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Project Foxhound

Server-Side

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Project Foxhound

Server-Side

- Less clear

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Project Foxhound

Server-Side

- Less clear
- SAST? DAST? Linter?

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Project Foxhound

Server-Side

- Less clear
- SAST? DAST? Linter?
- 💡: Investigate shared code

Detecting XSS

Client-Side

- Dynamic Taint Tracking

Project Foxhound

Server-Side

- Less clear
- SAST? DAST? Linter?
- 💡: Investigate shared code

⇒ Look at sanitizers!

Sanitization to Prevent XSS

- Simply remove or change dangerous parts from the input

Sanitization to Prevent XSS

- Simply remove or change dangerous parts from the input
  - Allow formatting tags to pass through, but remove everything dangerous
  - E.g., `<img src=x onerror=alert()>` → `<img src=x>`

Sanitization to Prevent XSS

- 💡 Simply remove or change dangerous parts from the input
  - Allow formatting tags to pass through, but remove everything dangerous
  - E.g., `<img src=x onerror=alert()>` → `<img src=x>`
- This is called **sanitization**

Sanitization to Prevent XSS

- Simply remove or change dangerous parts from the input
  - Allow formatting tags to pass through, but remove everything dangerous
  - E.g., `<img src=x onerror=alert()>` → `<img src=x>`
- This is called **sanitization**

Definition: Sanitizer

Function taking arbitrary input and returns a safe value

- The output shall resemble the input

  ⇒ I.e., perserve benign parts

My journey towards this research

- Researching people rolling their own sanitizers
- E.g., trying to filter HTML with regular expressions
  - I Spoke about this at RuhrSec in 2023

My journey towards this research

- Researching people rolling their own sanitizers
- E.g., trying to filter HTML with regular expressions
  - I Spoke about this at RuhrSec in 2023

```
function f(v) {
  return v.replace(/'/g, "").replace(/\(/g, "")
  .replace(/\)/g, "").replace(/alert/g, ""); }
```

How not to sanitize HTML

My journey towards this research

- Researching people rolling their own sanitizers
- E.g., trying to filter HTML with regular expressions
  - I Spoke about this at RuhrSec in 2023

```javascript
function f(v) {
  return v.replace(/'/g, "").replace(/\(/g, "")
  .replace(/\)/g, "").replace(/alert/g, ""); }
```

How not to sanitize HTML

- My takeaway: Use sanitizers relying on a real HTML parser
- I.e., most server-side sanitizers

My journey towards this research

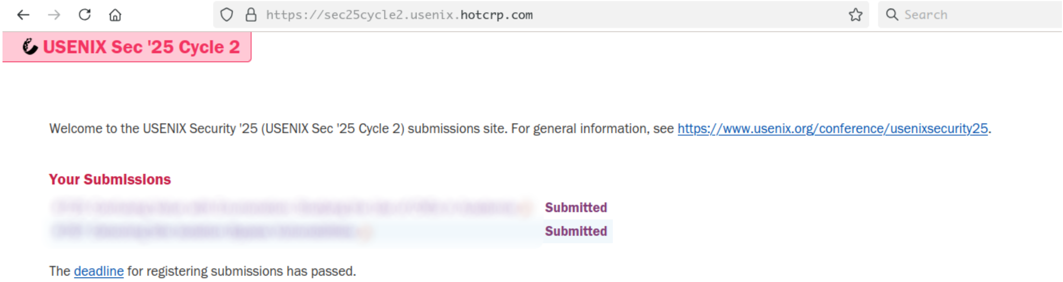
- Researching people rolling their own sanitizers
- E.g., trying to filter HTML with regular expressions
  - I Spoke about this at RuhrSec in 2023

```
function f(v) {
  return v.replace(/'/g, "").replace(/\(/g, "")
  .replace(/\)/g, "").replace(/alert/g, ""); }
```

How not to sanitize HTML

- My takeaway: Use sanitizers relying on a real HTML parser
- I.e., most server-side sanitizers
- But does that really help?

Example Application



USENIX Sec '25 Cycle 2

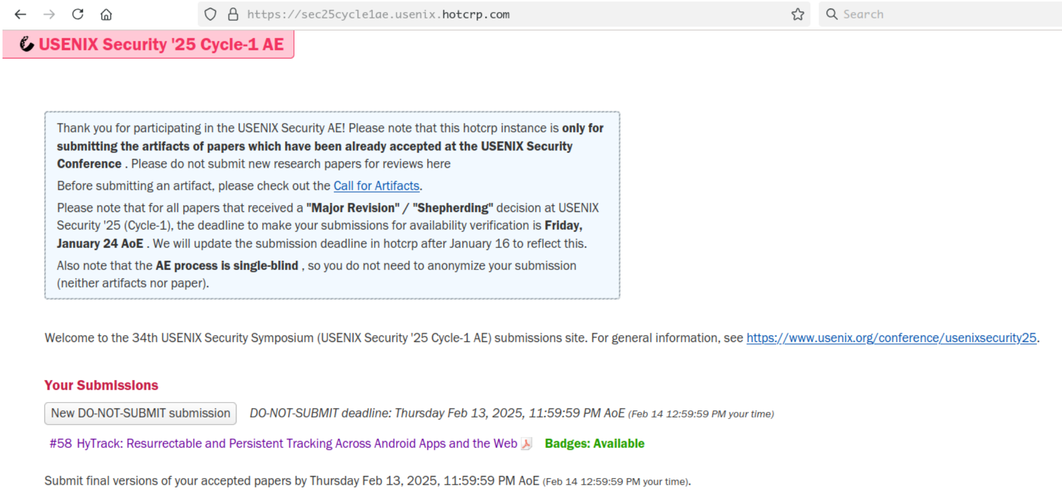
Welcome to the USENIX Security '25 (USENIX Sec '25 Cycle 2) submissions site. For general information, see https://www.usenix.org/conference/usenixsecurity25.

Your Submissions

[illegible]	Submitted
[illegible]	Submitted

The deadline for registering submissions has passed.

Example Application


https://sec25cycle1ae.usenix.hotcrp.com
Search
New DO-NOT-SUBMIT submission

USENIX Security '25 Cycle-1 AE

Thank you for participating in the USENIX Security AE! Please note that this hotcrp instance is **only for submitting the artifacts of papers which have been already accepted at the USENIX Security Conference** . Please do not submit new research papers for reviews here

Before submitting an artifact, please check out the Call for Artifacts.

Please note that for all papers that received a **"Major Revision" / "Shepherding"** decision at USENIX Security '25 (Cycle-1), the deadline to make your submissions for availability verification is **Friday, January 24 AoE** . We will update the submission deadline in hotcrp after January 16 to reflect this.

Also note that the **AE process is single-blind** , so you do not need to anonymize your submission (neither artifacts nor paper).

Welcome to the 34th USENIX Security Symposium (USENIX Security '25 Cycle-1 AE) submissions site. For general information, see https://www.usenix.org/conference/usenixsecurity25.

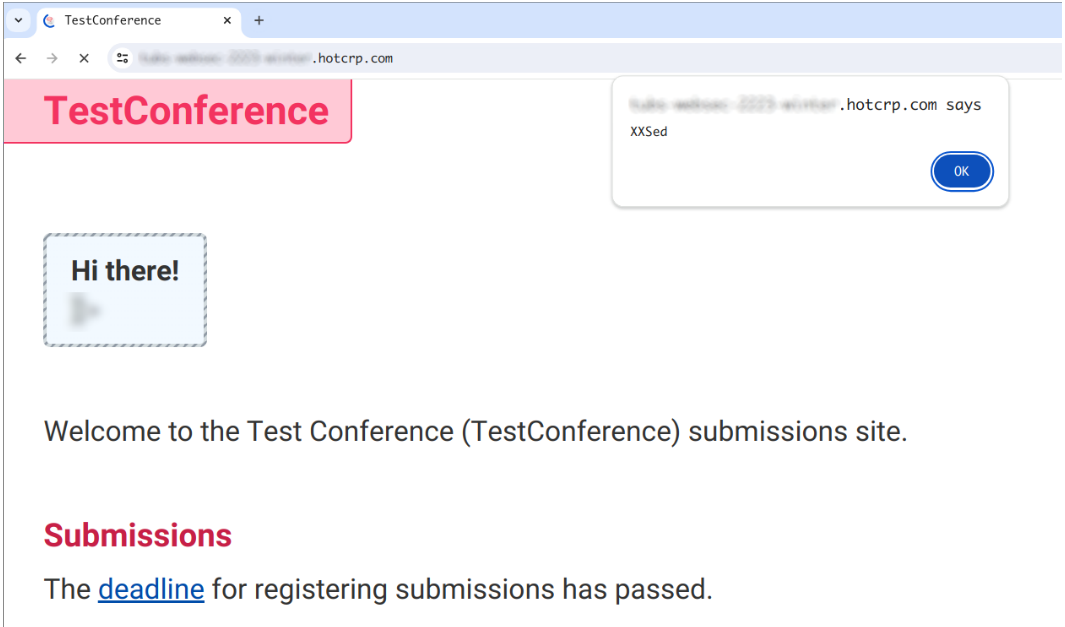
Your Submissions

DO-NOT-SUBMIT deadline: Thursday Feb 13, 2025, 11:59:59 PM AoE (Feb 14 12:59:59 PM your time)

#58 HyTrack: Resurrectable and Persistent Tracking Across Android Apps and the Web **Badges: Available**

Submit final versions of your accepted papers by Thursday Feb 13, 2025, 11:59:59 PM AoE (Feb 14 12:59:59 PM your time).

Secure? No!



[redacted].hotcrp.com says

XXSed

TestConference

Hi there!

Welcome to the Test Conference (TestConference) submissions site.

Submissions

The deadline for registering submissions has passed.

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

- E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, ...

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

- E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, . . .

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

- E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, ...

- By injecting custom JavaScript one can:

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

  - E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, . . .

- By injecting custom JavaScript one can:
  - Lure victim onto my conference page

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

  - E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, . . .

- By injecting custom JavaScript one can:
  - Lure victim onto my conference page
    - ▶ Simply spoof sender as `stock@cispa.de` or `director@cispa.de`

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

  - E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, …

- By injecting custom JavaScript one can:
  - Lure victim onto my conference page
    - Simply spoof sender as `stock@cispa.de` or `director@cispa.de`
  - Automatically log out visitor
  - Exfiltrate username and password on log in

Impact?

My test conference was hosted under `hotcrp.com`

⇒ Shares login data with all conferences on `hotcrp.com`

  - E.g., Usenix Security, CCS, NDSS, EuroS&P, RAID, …

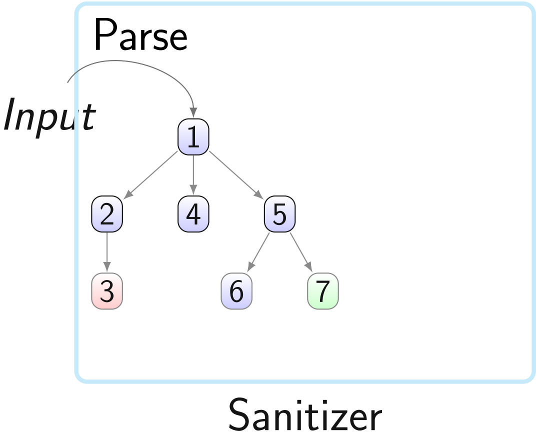
- By injecting custom JavaScript one can:
  - Lure victim onto my conference page
    - Simply spoof sender as `stock@cispa.de` or `director@cispa.de`
  - Automatically log out visitor
  - Exfiltrate username and password on log in

  ⇒ See everything they have access to

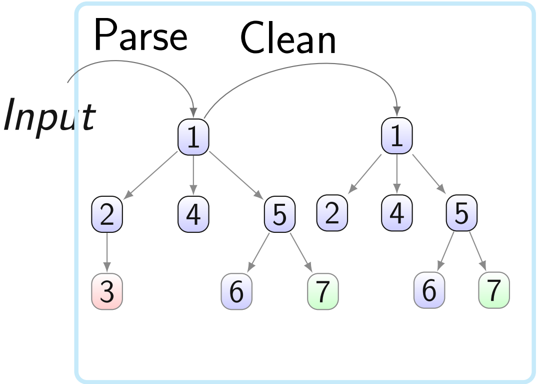
Sanitization: Workflow

Input

Sanitization: Workflow

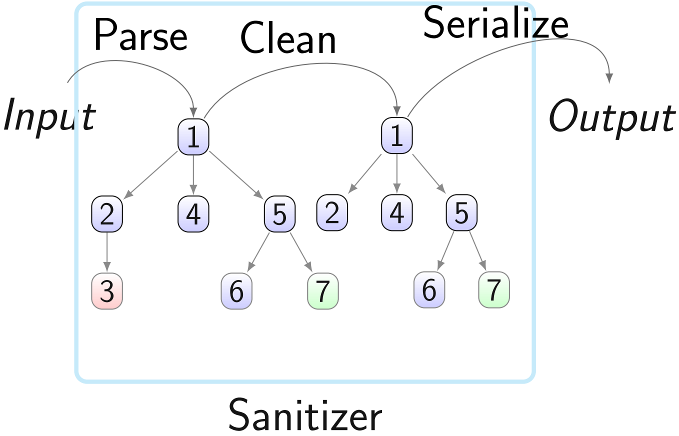


Sanitization: Workflow

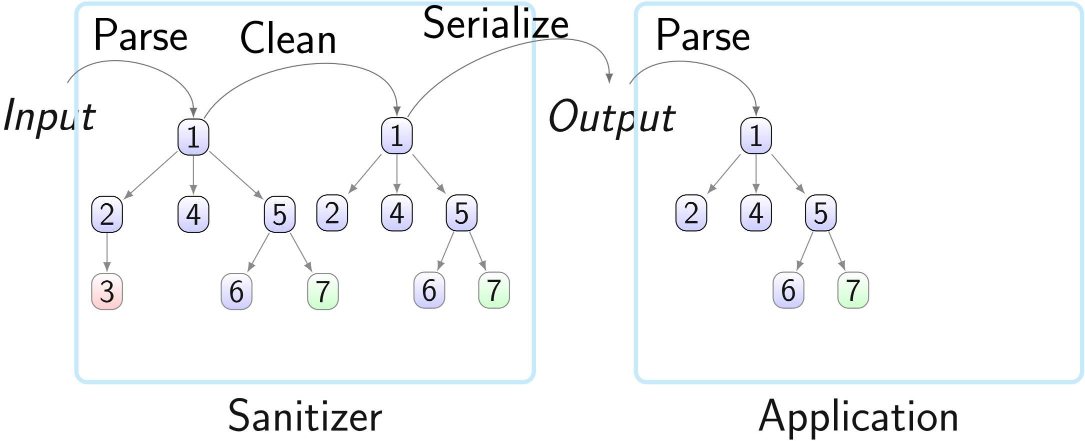


Sanitizer

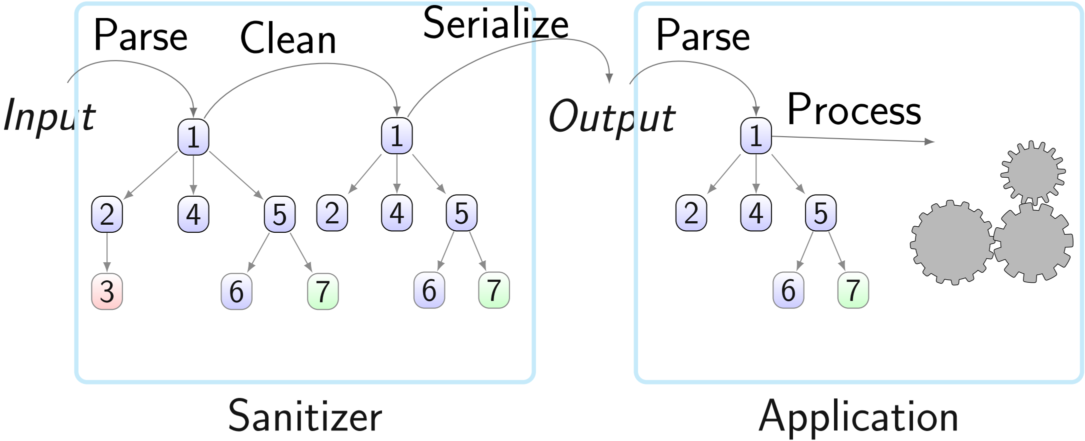
Sanitization: Workflow



Sanitization: Workflow



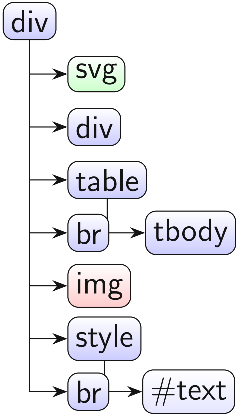
Sanitization: Workflow



HTML Parsing Complexities

HTML Code → Parsed into → **DOM Tree**

```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

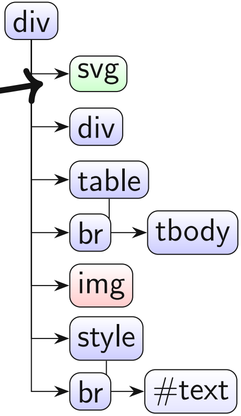


HTML Parsing Complexities

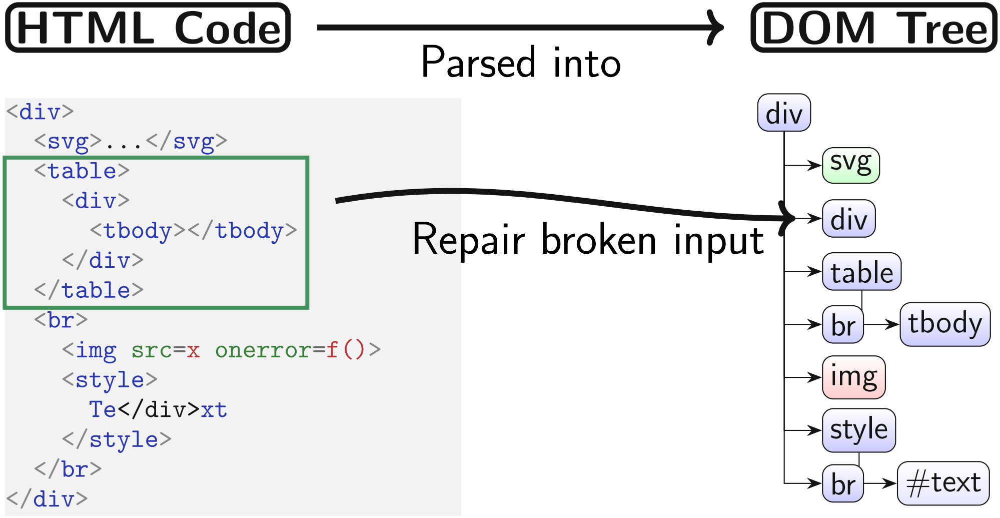
HTML Code → Parsed into → **DOM Tree**

```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

Change to SVG parser

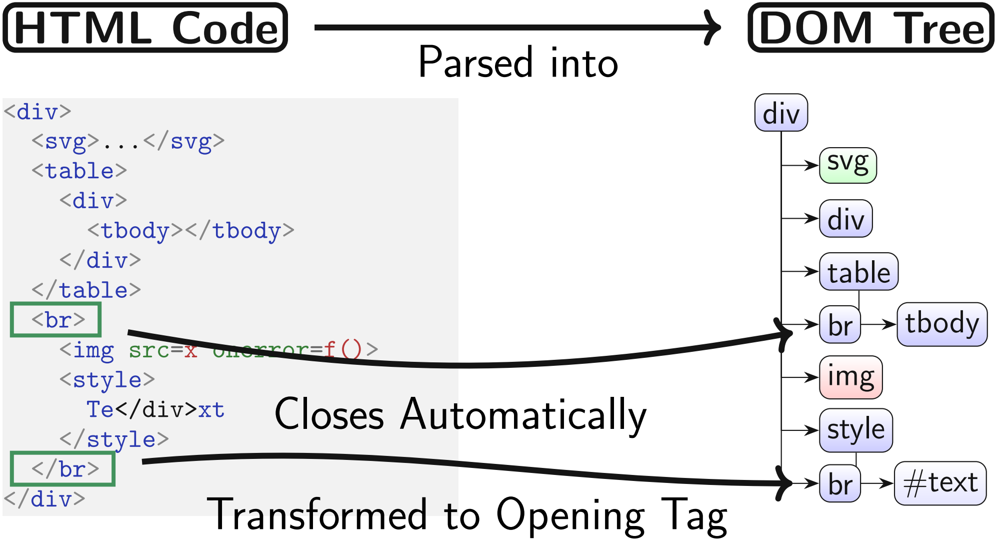


HTML Parsing Complexities



```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

HTML Parsing Complexities



HTML Code

Parsed into

DOM Tree

```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

Closes Automatically

Transformed to Opening Tag

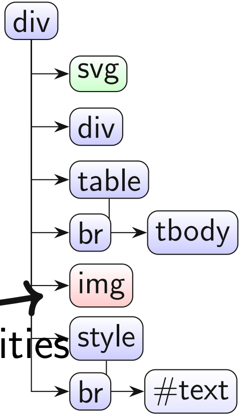
HTML Parsing Complexities

HTML Code → **DOM Tree**

Parsed into

```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

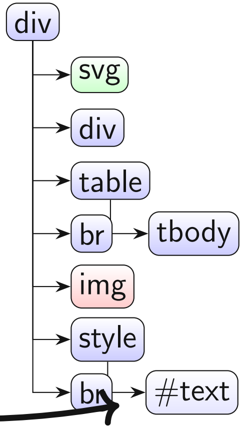
Script execution capabilities



HTML Parsing Complexities

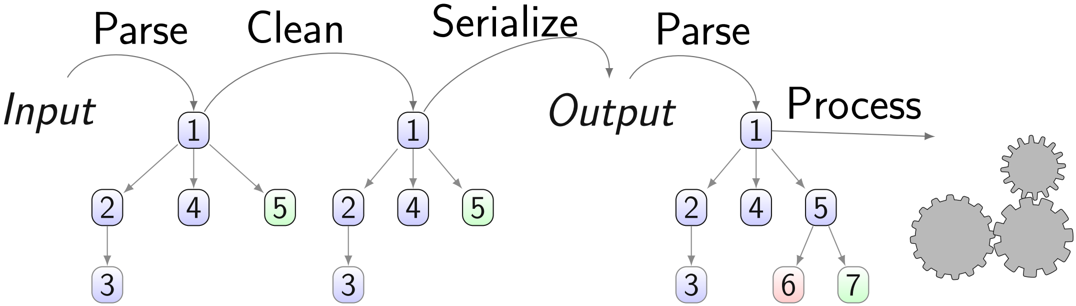
HTML Code → Parsed into → **DOM Tree**

```
<div>
  <svg>...</svg>
  <table>
    <div>
      <tbody></tbody>
    </div>
  </table>
  <br>
    <img src=x onerror=f()>
    <style>
      Te</div>xt
    </style>
  </br>
</div>
```

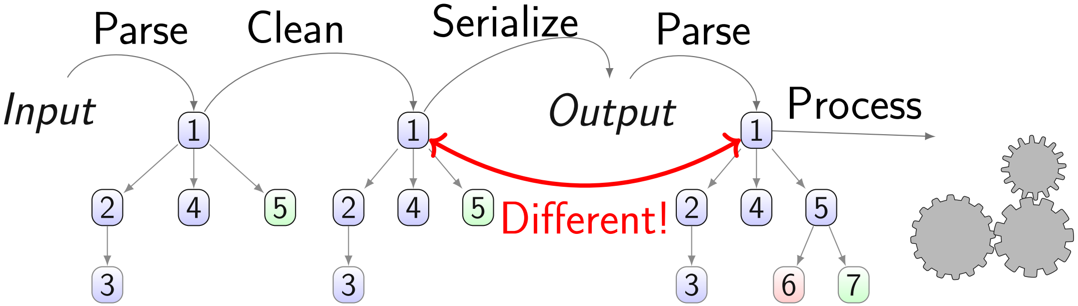


Different Parsing Mode

Sanitization: Parsing Differential



Sanitization: Parsing Differential



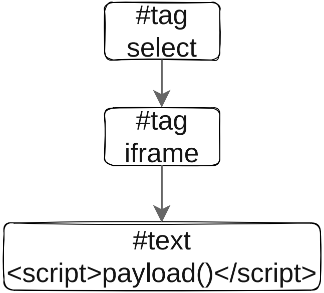
Parsing Differential to XSS

Payload: `<select><iframe><script>payload()</script>`

Parsing Differential to XSS

Payload: `<select><iframe><script>payload()</script>`

Parsed by Caja

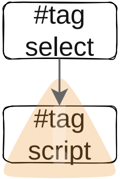


#tag
select

#tag
iframe

#text
<script>payload()</script>

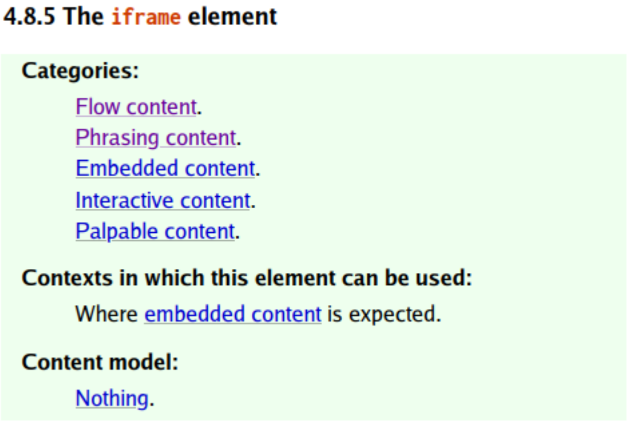
Parsed by Chrome



#tag
select

#tag
script

Root Cause



4.8.5 The iframe element

Categories:

Flow content.
Phrasing content.
Embedded content.
Interactive content.
Palpable content.

Contexts in which this element can be used:

Where embedded content is expected.

Content model:

Nothing.

Root Cause

The “nothing” content model:

. . . the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

Root Cause

> **The “nothing” content model:**
>
> …the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

- However, the parsing specification disagrees: content of `iframe` shall be parsed as text!

Root Cause

The “nothing” content model:

> . . . the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

- However, the parsing specification disagrees:

⇒ Inconsistency in the spec! One parsing quirk we identified

Root Cause

The “nothing” content model:

. . . the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

Results in `iframe` element with payload as textual content.

No code execution!

```
div.innerHTML = `<iframe><img src=x onerror=alert(1)>`;
```

Root Cause

The “nothing” content model:

> . . . the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

- However, the parsing specification disagrees:

⇒ Inconsistency in the spec! One parsing quirk we identified

- So the sanitizer is actually correct, but. . .

Root Cause

The "nothing" content model:

...the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

- However, the parsing specification disagrees:

⇒ Inconsistency in the spec! One parsing quirk we identified

- So the sanitizer is actually correct, but...

? Where has the `iframe` gone?

The Missing `iframe`

Recall the payload:

```
<select><iframe><script>payload()</script>
```

The Missing `iframe`

Recall the payload:

```
<select><iframe><script>payload()</script>
```

The `select` Element

Content model:

Zero or more `option`, `optgroup`, and script-supporting elements.

ℹ "script-supporting elements" are `script` and `template` tags

The Missing iframe

Recall the payload:

```
<select><iframe><script>payload()</script>
```

The select Element

Content model:

Zero or more `option`*,* `optgroup`*, and script-supporting elements.*

⇒ An iframe can't be a child of select!

- So Chrome simply drops it

Who Even Uses Google Caja?

- Google has deprecated Caja 5y+ ago

Who Even Uses Google Caja?

- Google has deprecated Caja 5y+ ago
- That does not stop others from using it, e.g.,:

Who Even Uses Google Caja?

- Google has deprecated Caja 5y+ ago
- That does not stop others from using it, e.g.,:
  - Adobe

Who Even Uses Google Caja?

- Google has deprecated Caja 5y+ ago
- That does not stop others from using it, e.g.,:
  - Adobe
  - SAP

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3
- CDATA is a SGML construct

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3
- CDATA is a SGML construct
  - `<![CDATA[<b> to emphasize]]>`

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3
- CDATA is a SGML construct
  - `<![CDATA[<b> to emphasize]]>`
- However, CDATA is not allowed in HTML!

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3
- CDATA is a SGML construct
  - `<![CDATA[<b> to emphasize]]>`
- However, CDATA is not allowed in HTML!

⇒ The Browser will fix it for you!

Hotcrp Parsing Differential

- Root cause: Handling CDATA sections
  - Same issue also affected Typo3
- CDATA is a SGML construct
  - `<![CDATA[<b> to emphasize]]>`
- However, CDATA is not allowed in HTML!

⇒ The Browser will fix it for you!

The parser treats such CDATA sections (including leading "[CDATA[" and trailing "]]" strings) as comments.

Hotcrp Parsing Differential

- `<![CDATA[a<b]]>` → `<!--[CDATA[a<b]]-->`

Hotcrp Parsing Differential

- `<![CDATA[a<b]]>` → `<!--[CDATA[a<b]]-->`
- However, if the CDATA section contains >:

Hotcrp Parsing Differential

- `<![CDATA[a<b]]>` → `<!--[CDATA[a<b]]-->`
- However, if the CDATA section contains >:
- `<![CDATA[<b><t>]]>` → `<!--[CDATA[<b--><t>]]&gt;`

Hotcrp Parsing Differential

- `<![CDATA[a<b]]>` → `<!--[CDATA[a<b]]-->`
- However, if the CDATA section contains >:
- `<![CDATA[<b><t>]]>` → `<!--[CDATA[<b--><t>]]&gt;`
- `<![CDATA[<b><img src=x onerror=f()>]]>` → `<!--[CDATA[<b--><img src=x onerror=f()>]]&gt;`

MutaGen

- Goal: Find Parsing Differentials to bypass HTML sanitizers

MutaGen

- Goal: Find Parsing Differentials to bypass HTML sanitizers

MutaGen: HTML payload generator

💡 Generate HTML that is difficult to parse

MutaGen

- Goal: Find Parsing Differentials to bypass HTML sanitizers

MutaGen: HTML payload generator

💡 Generate HTML that is difficult to parse

⇒ It *mutates* during parsing

`MutaGen`

- Goal: Find Parsing Differentials to bypass HTML sanitizers

`MutaGen`: HTML payload generator

💡 Generate HTML that is difficult to parse

⇒ It *mutates* during parsing

- Important to keep in mind: HTML parsing never fails!

MutaGen

- Goal: Find Parsing Differentials to bypass HTML sanitizers

MutaGen: HTML payload generator

💡 Generate HTML that is difficult to parse

⇒ It *mutates* during parsing

- Important to keep in mind: HTML parsing never fails!

⇒ Garbage in, DOM out

MutaGen: Payload Generation

Generation

Serialization

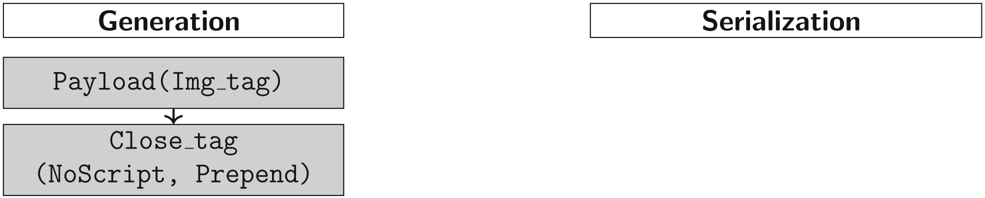
MutaGen: Payload Generation

Generation

`Payload(Img_tag)`

Serialization

MutaGen: Payload Generation



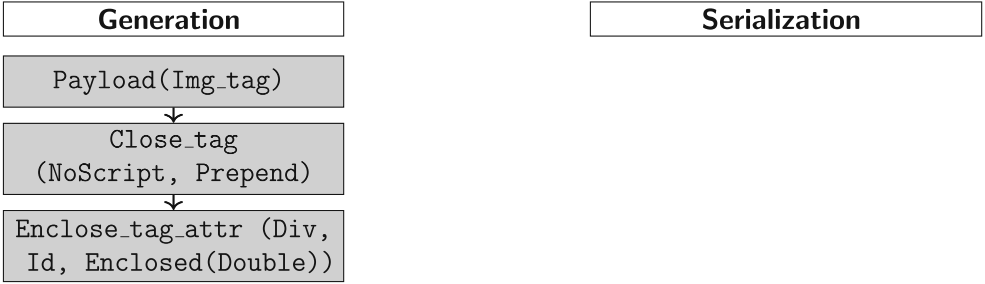
Generation

Serialization

Payload(Img_tag)

Close_tag (NoScript, Prepend)

MutaGen: Payload Generation



Generation

`Payload(Img_tag)`

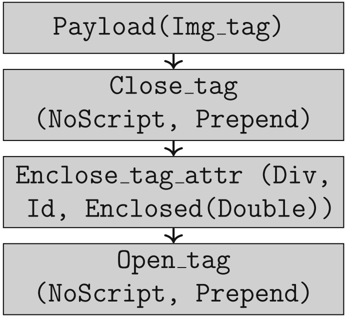
`Close_tag (NoScript, Prepend)`

`Enclose_tag_attr (Div, Id, Enclosed(Double))`

Serialization

MutaGen: Payload Generation

Generation



Payload(Img_tag)

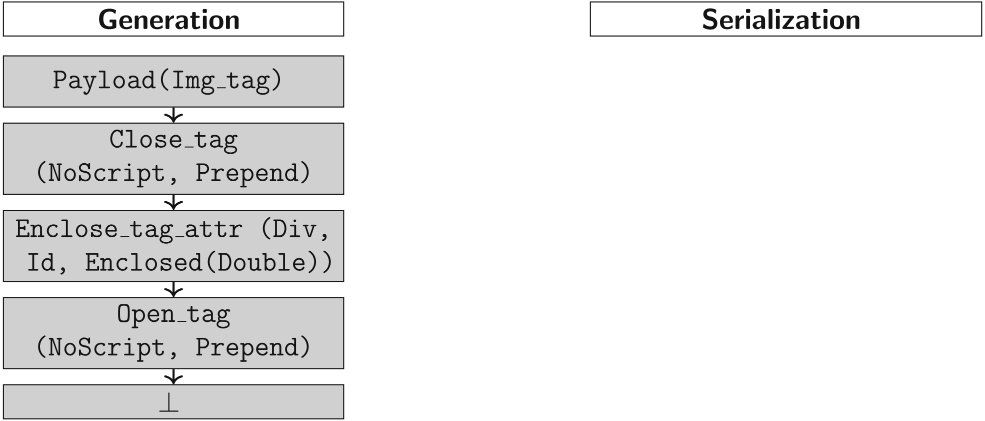
Close_tag (NoScript, Prepend)

Enclose_tag_attr (Div, Id, Enclosed(Double))

Open_tag (NoScript, Prepend)

Serialization

MutaGen: Payload Generation



Generation

Serialization

Payload(Img_tag)

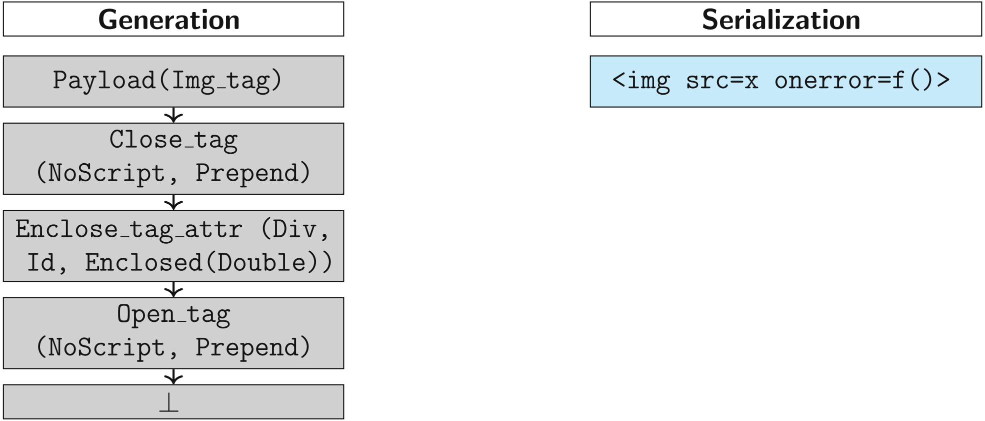
Close_tag (NoScript, Prepend)

Enclose_tag_attr (Div, Id, Enclosed(Double))

Open_tag (NoScript, Prepend)

⊥

MutaGen: Payload Generation



Generation

Payload(Img_tag)

Close_tag (NoScript, Prepend)

Enclose_tag_attr (Div, Id, Enclosed(Double))

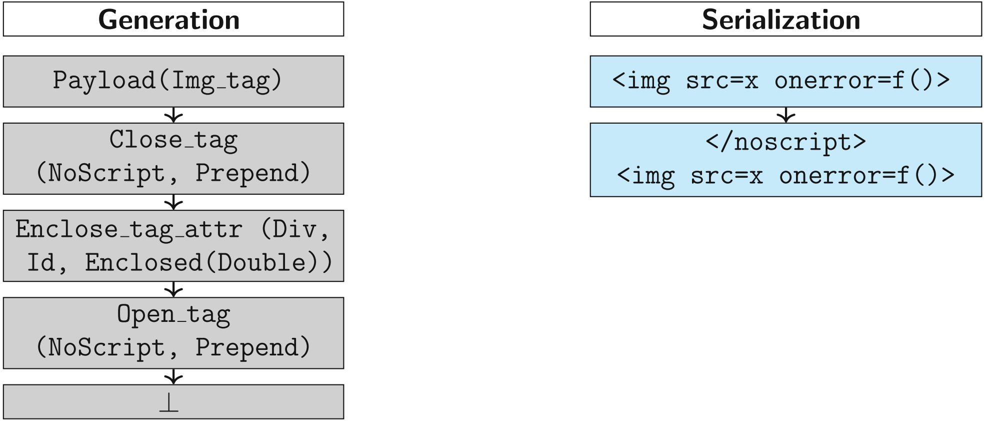
Open_tag (NoScript, Prepend)

⊥

Serialization

`<img src=x onerror=f()>`

MutaGen: Payload Generation



Generation

`Payload(Img_tag)`

`Close_tag (NoScript, Prepend)`

`Enclose_tag_attr (Div, Id, Enclosed(Double))`

`Open_tag (NoScript, Prepend)`

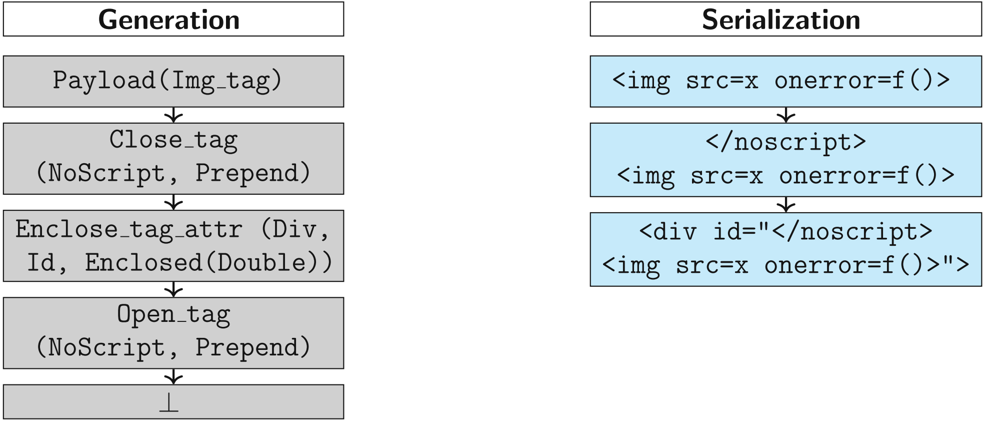
$\perp$

Serialization

`<img src=x onerror=f()>`

`</noscript> <img src=x onerror=f()>`

MutaGen: Payload Generation



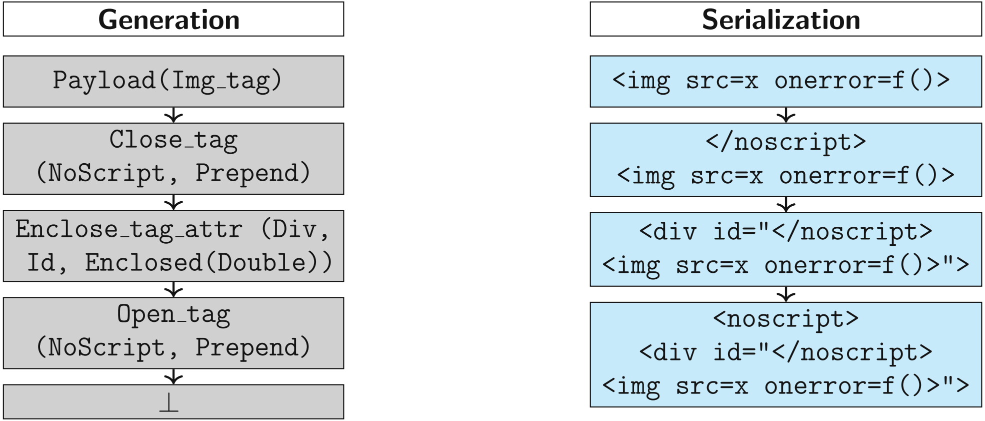
Generation

```
Payload(Img_tag)
↓
Close_tag
(NoScript, Prepend)
↓
Enclose_tag_attr (Div,
 Id, Enclosed(Double))
↓
Open_tag
(NoScript, Prepend)
↓
⊥
```

Serialization

```
<img src=x onerror=f()>
↓
</noscript>
<img src=x onerror=f()>
↓
<div id="</noscript>
<img src=x onerror=f()>">
```

MutaGen: Payload Generation



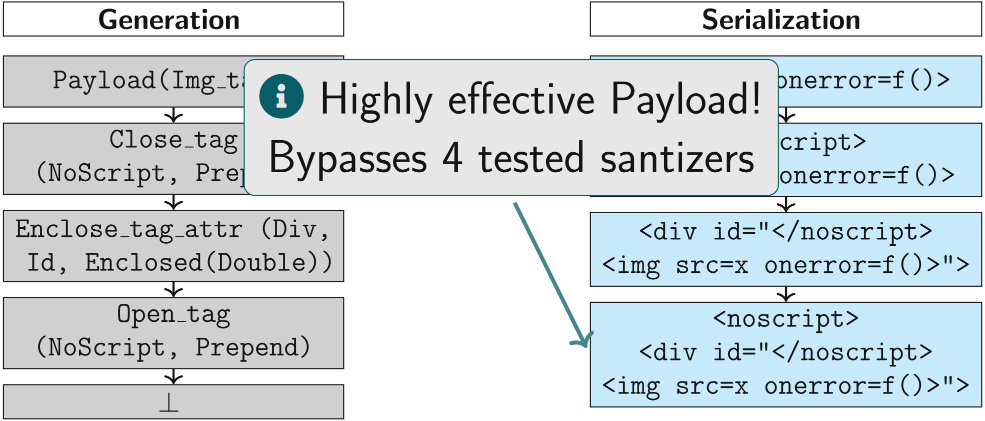
Generation

```
Payload(Img_tag)
↓
Close_tag
(NoScript, Prepend)
↓
Enclose_tag_attr (Div,
 Id, Enclosed(Double))
↓
Open_tag
(NoScript, Prepend)
↓
⊥
```

Serialization

```
<img src=x onerror=f()>
↓
</noscript>
<img src=x onerror=f()>
↓
<div id="</noscript>
<img src=x onerror=f()>">
↓
<noscript>
<div id="</noscript>
<img src=x onerror=f()>">
```

MutaGen: Payload Generation



Generation

Payload(Img_ta

Close_tag
(NoScript, Pre

Enclose_tag_attr (Div,
Id, Enclosed(Double))

Open_tag
(NoScript, Prepend)

⊥

Serialization

onerror=f()>

cript>
onerror=f()>

<div id="</noscript>
<img src=x onerror=f()>">

<noscript>
<div id="</noscript>
<img src=x onerror=f()>">

ℹ Highly effective Payload!

Bypasses 4 tested santizers

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET

Parsing Differentials in the Wild

Name	Total Downloads	Language	Vulns.
DOMPurify	399 001 216	JavaScript	**2**
google caja	41 305 997		*x*
sanitize-html	276 882 692		**0**
HtmlSanitizer	19 800 000	.NET	**2**
HtmlRuleSanitizer	306 100		**2**
Typo3 html-sanitizer	1 950 185	PHP	**4**
rgrove/sanitize	60 928 006	Ruby	**1**
loofah	396 621 861		**0**
AntiSamy	No data available	Java	**3**
JSoup			**2**
Total	Over 1 Billion		**16**

Running `MutaGen`

During the first test, after like 10s, I was greeted by:

```
PHP Warning: Uninitialized string offset
26 in html5/src/HTML5/Parser/Scanner.php
on line 108
```

Running MutaGen

During the first test, after like 10s, I was greeted by:

```
PHP Warning: Uninitialized string offset
26 in html5/src/HTML5/Parser/Scanner.php
on line 108
```

A target nobody has fuzzed before, i.e., good target!

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET
- **All** have functional deficiencies

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET
- **All** have functional deficiencies
  - Average parsing similarity compared to browsers is below 60%

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET

- **All** have functional deficiencies
  - Average parsing similarity compared to browsers is below 60%
  - Even if secure, sanitizers mangle input by parsing incorrectly

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET

- **All** have functional deficiencies
  - Average parsing similarity compared to browsers is below 60%
  - Even if secure, sanitizers mangle input by parsing incorrectly
- 16 new bypass vectors across 9 of them

Parsing Differentials in the Wild

⇒ 11 sanitizers across five programming languages.

- Java, JavaScript, PHP, Ruby, and .NET

- **All** have functional deficiencies
  - Average parsing similarity compared to browsers is below 60%
  - Even if secure, sanitizers mangle input by parsing incorrectly
- 16 new bypass vectors across 9 of them
  - And one bypass vector in a sanitizer not directly tested by us

Parsing Accuracy #2

What parser processes the output? Fragment or Document?

Parsing Accuracy #2

What parser processes the output? Fragment or Document?

I.e., `innerHTML` assignment or `document.write`

Parsing Accuracy #2

What parser processes the output? Fragment or Document?

I.e., `innerHTML` assignment or `document.write`

Which browser is the result displayed in?

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```

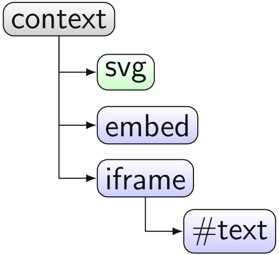
Does this execute code?

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```

Browser Parsing Differentials

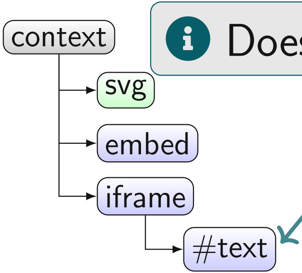
```
<svg><embed><iframe><desc><img src=x onerror=f()>
```



Chrome parsing result

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```

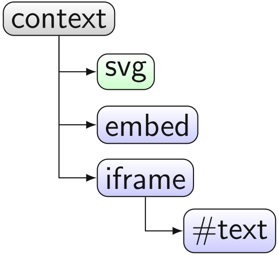


Chrome parsing result

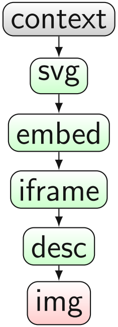
ℹ Does Not Execute Code!

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```



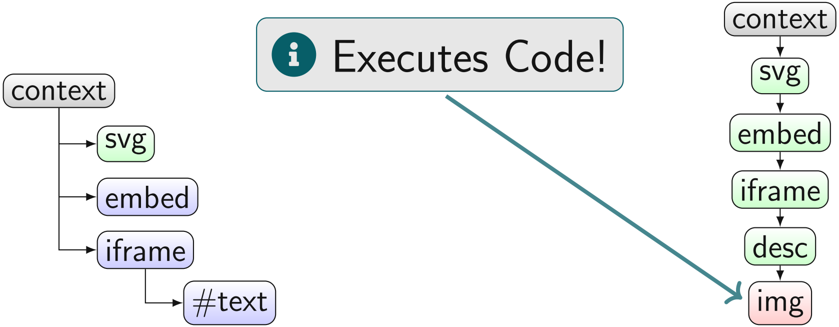
Chrome parsing result



Firefox parsing result

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```



Executes Code!

context
svg
embed
iframe
#text

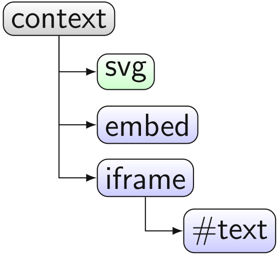
context
svg
embed
iframe
desc
img

Chrome parsing result

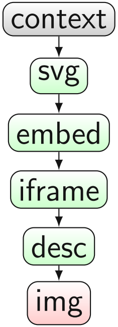
Firefox parsing result

Browser Parsing Differentials

```
<svg><embed><iframe><desc><img src=x onerror=f()>
```



Chrome parsing result



Firefox parsing result

⇒ Perfectly accurate sanitizer is impossible

DOMPurify to Aid Exploitation

Input: `<svg><style>&lt;img src=x onerror=f()&gt;<keygen>`

DOMPurify to Aid Exploitation

Input: `<svg><style>&lt;img src=x onerror=f()&gt;<keygen>`

Output: `<svg><style><img src=x onerror=f()>`

DOMPurify to Aid Exploitation

Input: `<svg><style>&lt;img src=x onerror=f()&gt;<keygen>`

Output: `<svg><style><img src=x onerror=f()>`

⇒ Sanitizers can help to bypass other security measures!

Common Problems

- Handling comments is surprisingly error prone. . .

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*

ℹ That is, comments terminated with `--!>`

Common Problems

- Handling comments is surprisingly error prone...
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses
  - Parsing depends on internal browser state, not exposed to sanitizers

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses
  - Parsing depends on internal browser state, not exposed to sanitizers

ℹ Sanitizing inputs containing `noscript` impossible!

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses
  - Parsing depends on internal browser state, not exposed to sanitizers
- Namespace confusion bugs are common

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses
  - Parsing depends on internal browser state, not exposed to sanitizers
- Namespace confusion bugs are common

ℹ Not correctly switching between different parsers.
Recall the Firefox bug shown earlier!

Common Problems

- Handling comments is surprisingly error prone. . .
  - Three sanitizers do not detect *closing bang comments*
- `noscript` is impossible to get right: four bypasses
  - Parsing depends on internal browser state, not exposed to sanitizers
- Namespace confusion bugs are common
- Some fundamental parsing bugs too!

Closing

Contact

✉ david.klein@tu-braunschweig.de

leinea

twitter.com/ncd_leen

Resources

ias-tubs/HTML_parsing_differentials

sap/project-foxhound

Main Takeaways

Parse → Serialize → Parse is prone to parsing differentials

Main Takeaways

Parse → Serialize → Parse is prone to parsing differentials

Server-Side HTML Sanitization is Insecure, Broken or Both

Main Takeaways

Parse → Serialize → Parse is prone to parsing differentials

Server-Side HTML Sanitization is Insecure, Broken or Both

A New Vision of Sanitization is Required to Get us Out of This Mess