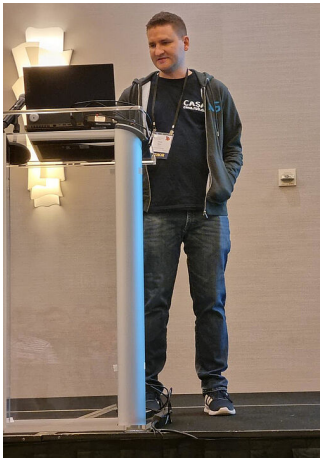

# Parse Me Baby One More Time: Bypassing HTML Sanitizer via Parsing Differentials

Speaker: David Klein

# About Me



- PhD Candidate
- Research interests:
  - Web Security
  - Privacy
  - Application Security

IAS | INSTITUTE FOR APPLICATION SECURITY

CASA
CYBER SECURITY IN THE AGE OF LARGE-SCALE ADVERSARIES

CAROLO-WILHELMINA BRAUNSCHWEIG
Technische Universität Braunschweig

# 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)

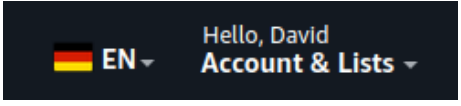
## Client-Side

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

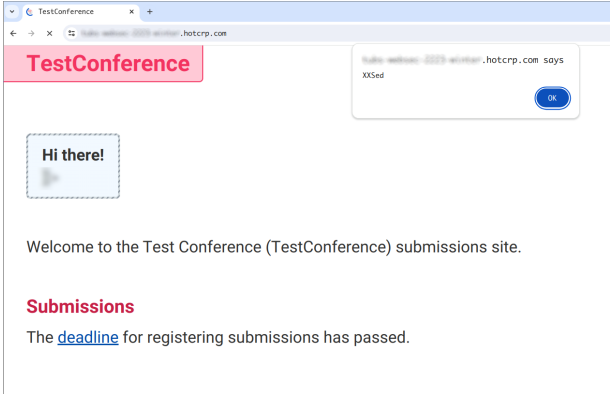
## Server-Side

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

## Such Code Patterns Are Everywhere!



# Everywhere?


TestConference
.hotcrp.com
TestConference
.hotcrp.com says
XXSed
OK
Hi there!
Welcome to the Test Conference (TestConference) submissions site.
Submissions
The deadline for registering submissions has passed.

# Detecting XSS

## Client-Side

- Dynamic Taint Tracking!
  - A taint browser

Project Foxhound

## Server-Side

- Less clear
- SAST? DAST? Linter?

# 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

```
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

```
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

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

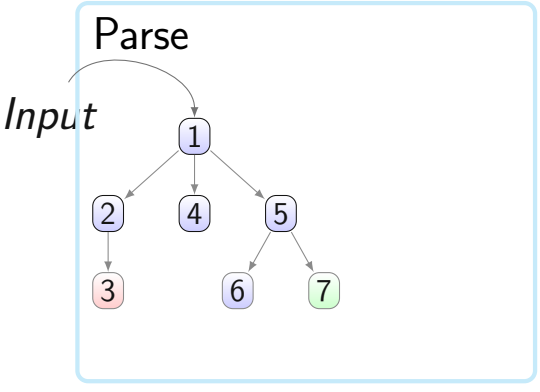
How not to sanitize HTML

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

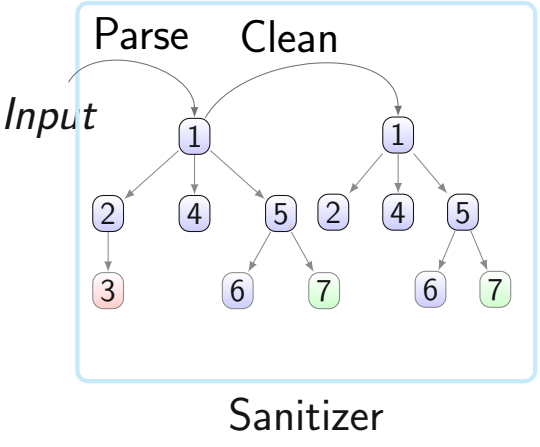
# Sanitization: Workflow

*Input*

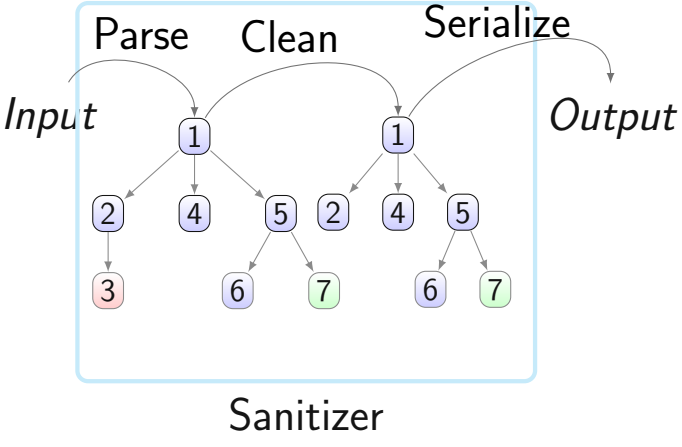
# Sanitization: Workflow



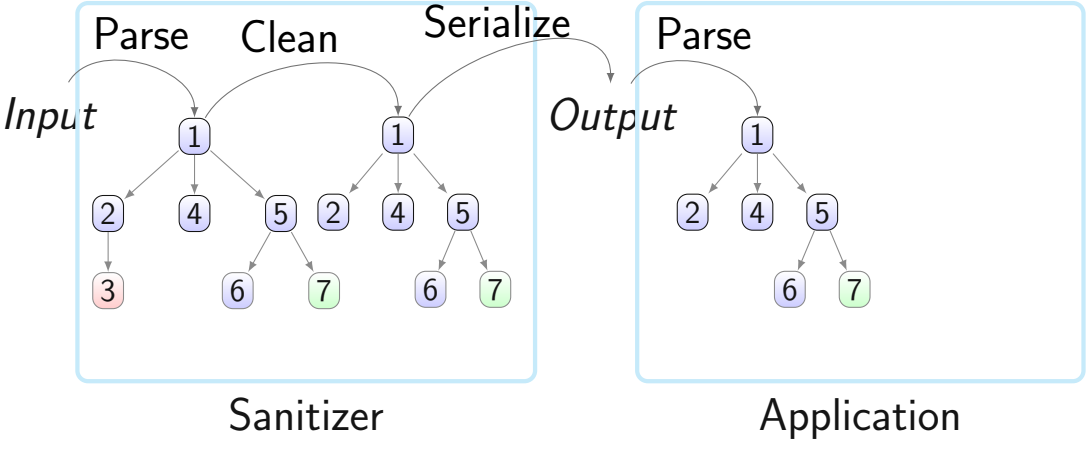
# Sanitization: Workflow



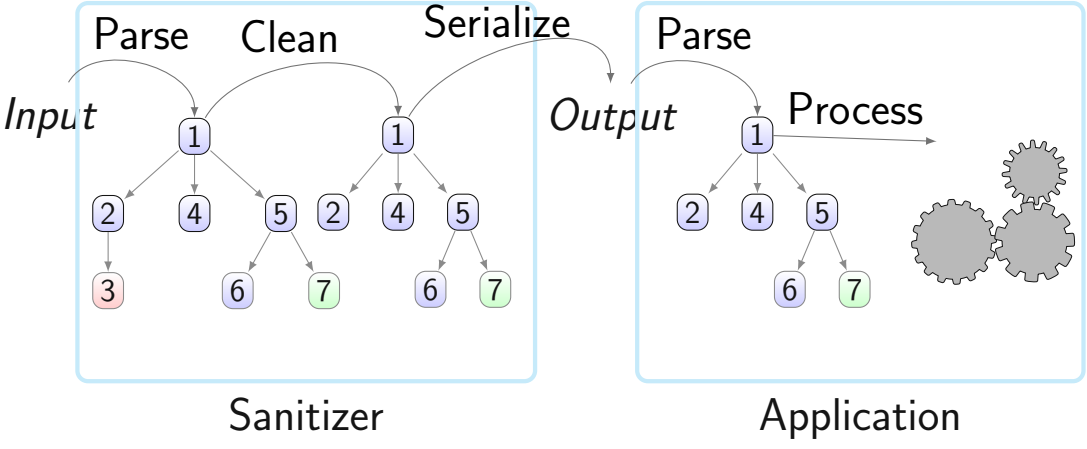
# Sanitization: Workflow



# Sanitization: Workflow



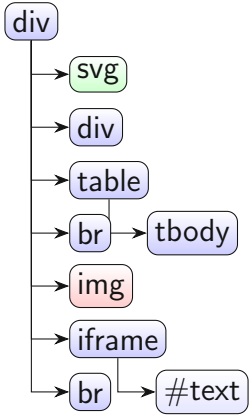
# Sanitization: Workflow



# 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>
```

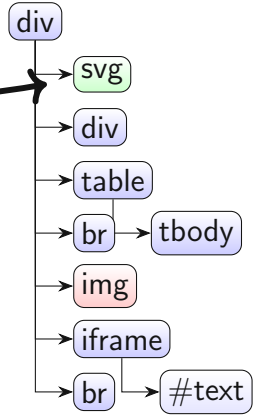


# HTML Parsing Complexities

**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

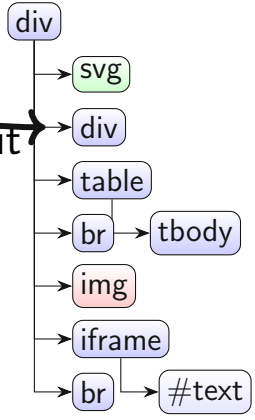


# HTML Parsing Complexities

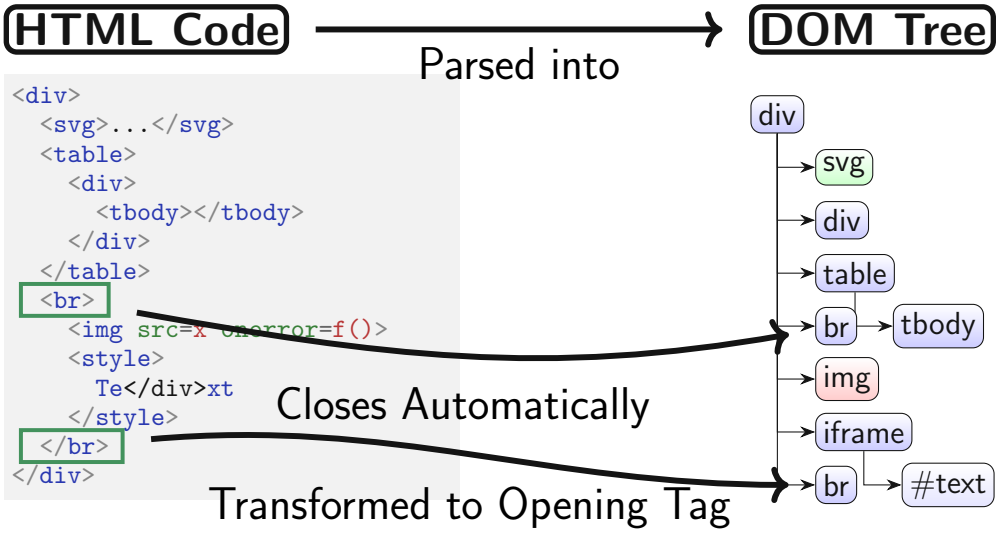
**HTML Code** → Parsed into → **DOM Tree**

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

Repair broken input



# HTML Parsing Complexities

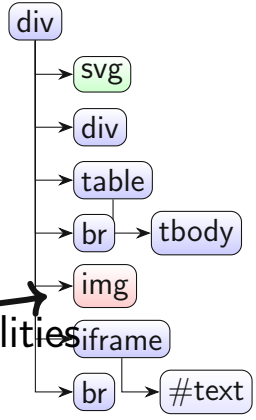


# HTML Parsing Complexities

**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>
```

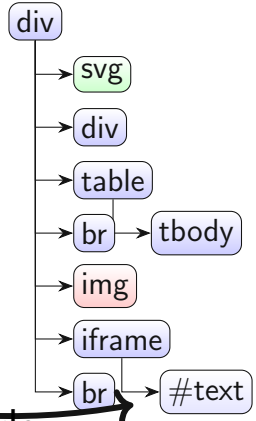
Script execution capabilities



# HTML Parsing Complexities

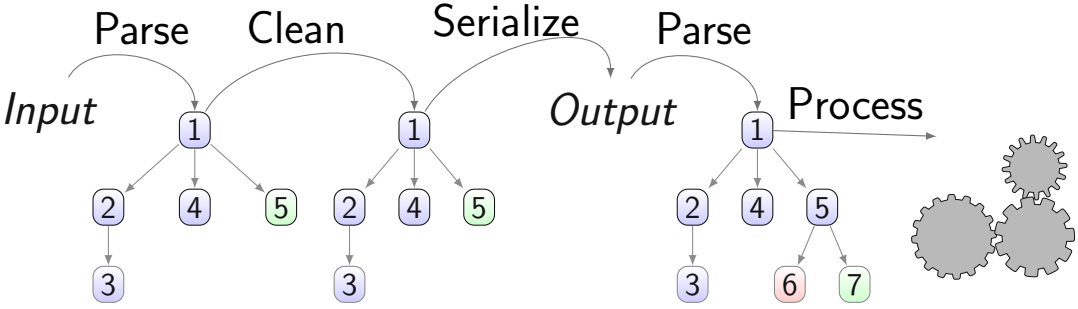
**HTML Code** → Parsed into → **DOM Tree**

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

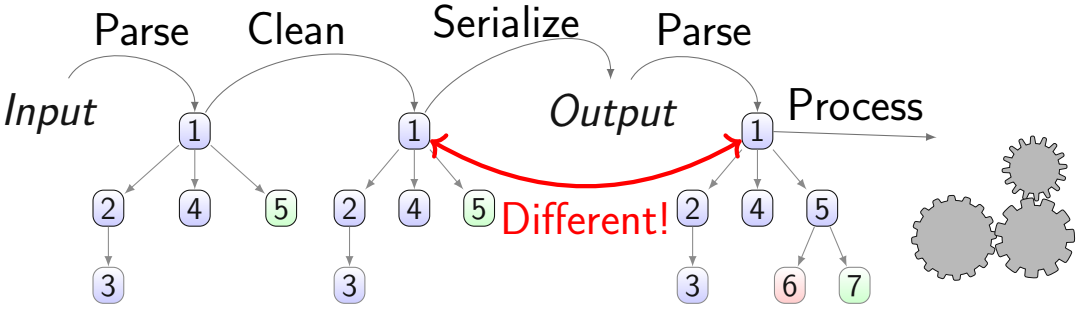



Different Parsing Mode

# Sanitization: Parsing Differential



# Sanitization: Parsing Differential



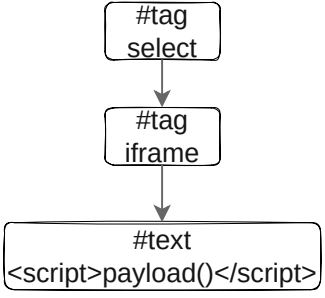
# Parsing Differential to XSS

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

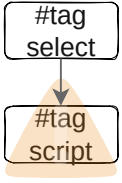
# Parsing Differential to XSS

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

## Parsed by Caja




## Parsed by Chrome



# 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`

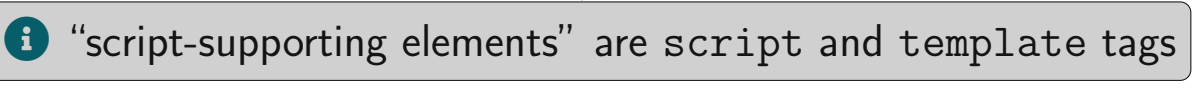
Recall the payload:

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

**The `select` Element**

**Content model:**

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



# 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 Uses Google Caja?

- Google has deprecated Caja 5y+ ago
- That does not stop others from using it, however
- 

- 

# 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

$\Rightarrow$ 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!

⇒ Garbage in, DOM out

Simplified Payload Generation and Serialization.

**Generation**

**Serialization**

Simplified Payload Generation and Serialization.

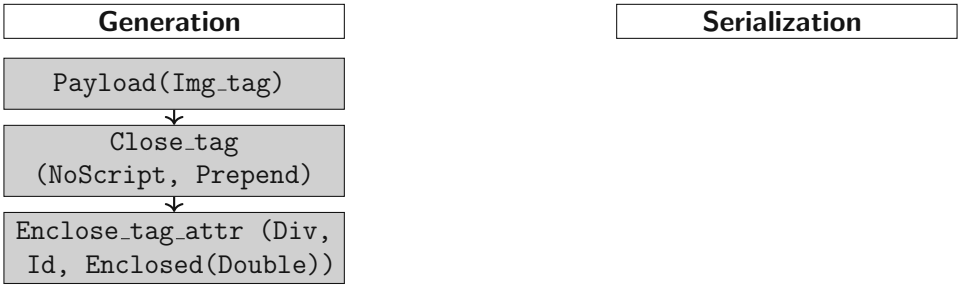
**Generation**

```
Payload(Img_tag)
```

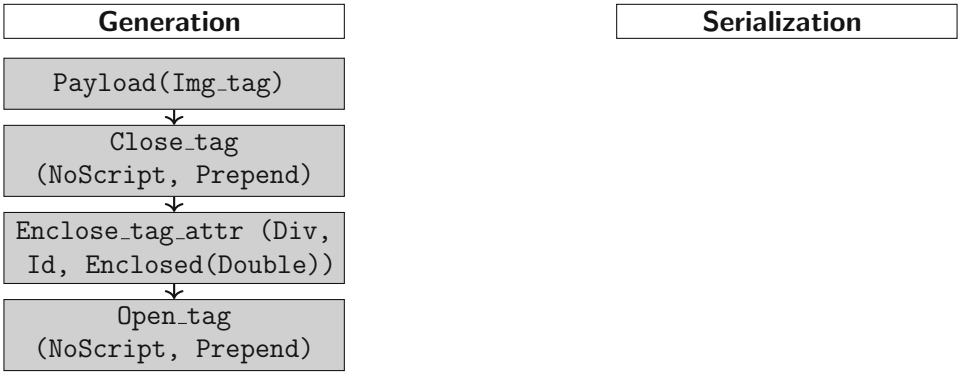
**Serialization**

Simplified Payload Generation and Serialization.

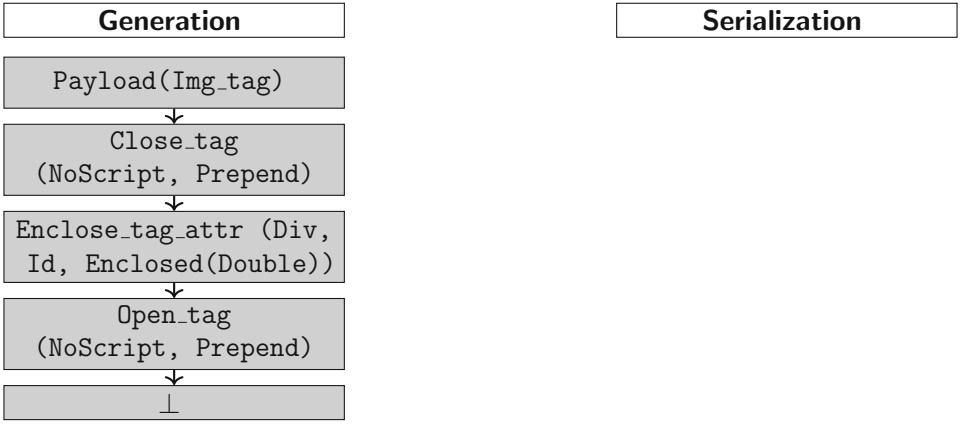
Simplified Payload Generation and Serialization.



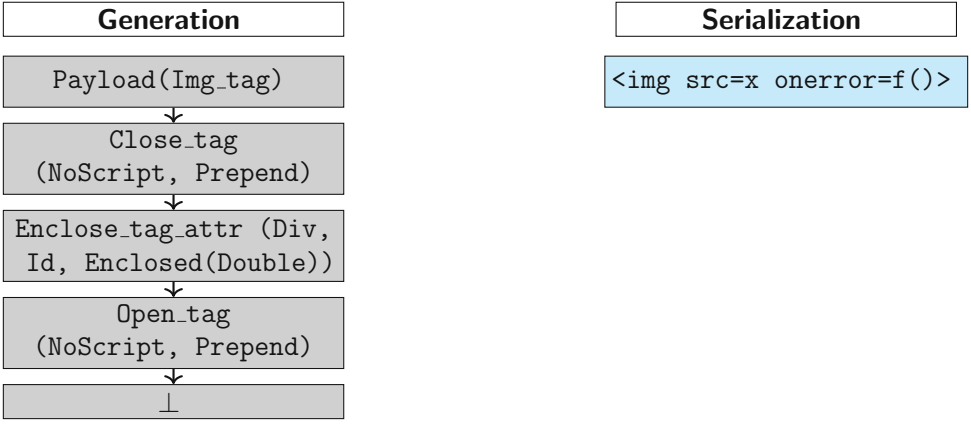
Simplified Payload Generation and Serialization.



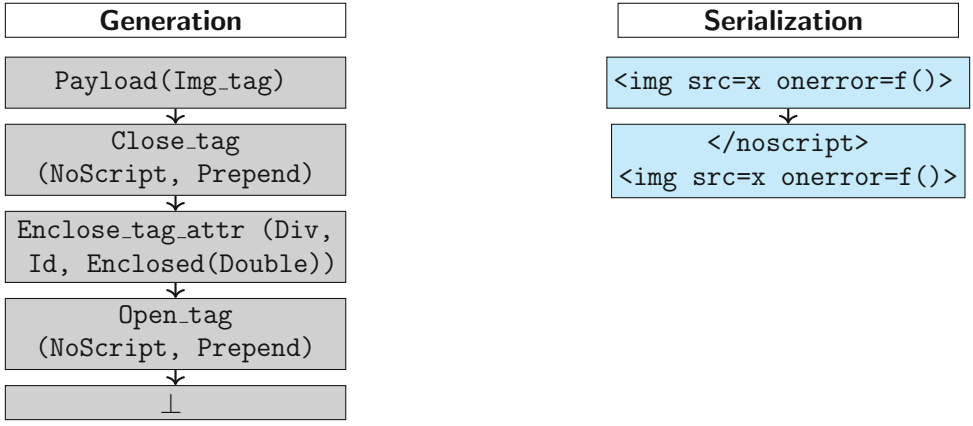
Simplified Payload Generation and Serialization.



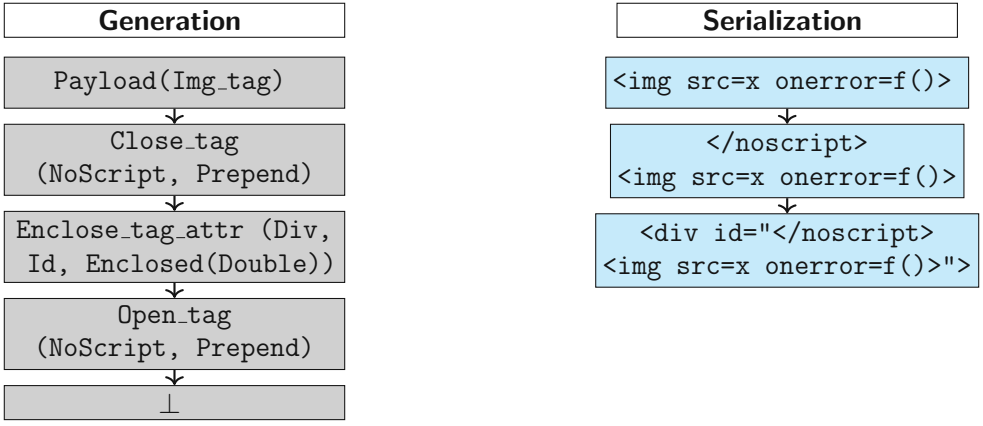
Simplified Payload Generation and Serialization.



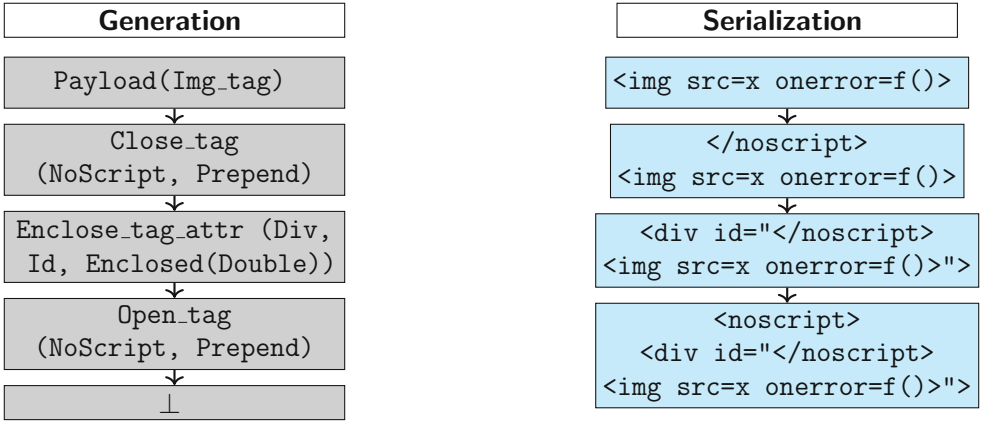
Simplified Payload Generation and Serialization.



Simplified Payload Generation and Serialization.



Simplified Payload Generation and Serialization.



# Parsing Differential in the Wild

⇒ 11 sanitizers across five programming languages.

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

# Parsing Differential in the Wild

| Name | Total Downloads | Language | Vulns. |
|---|---|---|---|
| DOMPurify | 399 001 216 | JavaScript | **2** |
| google caja | 41 305 997 | | † |
| 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
```

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

# Parsing Differential in the Wild

⇒ 11 sanitizers across five programming languages.

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

Payload:

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

# Browser Parsing Differentials

Payload:

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

**Does this execute code?**

# Browser Parsing Differentials

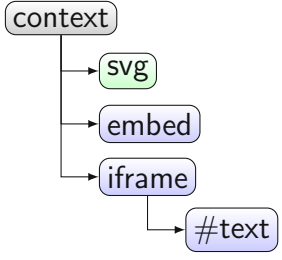
Payload:

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

# Browser Parsing Differentials

Payload:

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

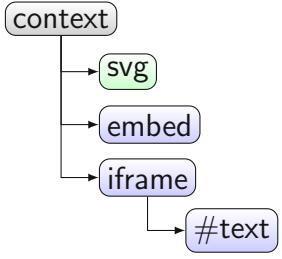



(a) Chrome parsing result

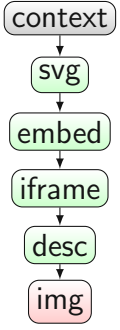
# Browser Parsing Differentials

Payload:

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




(a) Chrome parsing result




(b) 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

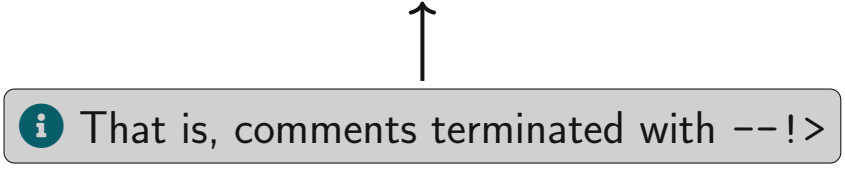
- Handling comments is surprisingly error prone...

# 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*



# 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 browser internal 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 browser internal state, not exposed to sanitizers

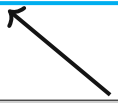
Sanitizing inputs containing `noscript` impossible!

# 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 browser internal 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 browser internal state, not exposed to sanitizers
- Namespace confusion bugs are common



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 browser internal state, not exposed to sanitizers
- Namespace confusion bugs are common
- Some fundamental parsing bugs too!

# Thank you!

## If you want to chat Web Security please get in touch!

### Contact

david.klein@tu-braunschweig.de
leinea
twitter.com/ncd_leen

# Black Hat Sound Bytes

**Server-Side HTML Sanitization is Insecure, Broken or Both**

**Parse $\rightarrow$ Serialize $\rightarrow$ Parse is always prone to parsing differentials**

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