Hand Sanitizers in the Wild

A Large-scale Study of Custom JavaScript Sanitizer Functions

David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, Martin Johns

Institute for Application Security
Technische Universität Braunschweig

david.klein@tu-braunschweig.de

Motivation

Client Side XSS: Root Cause

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4     Application code
5  */
6  div.innerHTML = greeting;
```

Client Side XSS: Root Cause

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4     Application code
5  */
6  div.innerHTML = greeting;
```

Visiting:
foo.com#RuhrSec

Hello, RuhrSec

Client Side XSS: Root Cause

```
1 let name = location.hash.substr(1);
2 let greeting = "Hello, " + name;
3 /*
4    Application code
5 */
6 div.innerHTML = greeting;
```

Visiting:
`foo.com#<b>RuhrSec</b>`

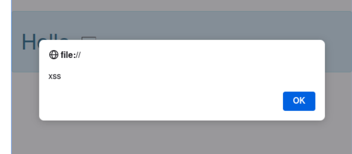
Hello, **RuhrSec**

Client Side XSS: Root Cause

```
1 let name = location.hash.substr(1);
2 let greeting = "Hello, " + name;
3 /*
4    Application code
5 */
6 div.innerHTML = greeting;
```

Visiting:

`foo.com#<img src=x onerror=alert('xss')>`



Client Side XSS: Root Cause

No difference between data and markup in HTML

```
1 let name = location.hash.substr(1);
2 let greeting = "Hello, " + name;
3 /*
4    Application code
5 */
6 div.innerHTML = greeting;
```

Client Side XSS: Root Cause

Source: Attacker controlled data

Source

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4     Application code
5  */
6  div.innerHTML = greeting;
```

Client Side XSS: Root Cause

Sink: Turned into (executable) code

```
let name = location.hash.substr(1);
let greeting = "Hello, " + name;
/*
   Application code
*/
div.innerHTML = greeting;
```

Sink

Client Side XSS: Root Cause

Unprotected data flow from source to sink

Source

```
1 let name = location.hash.substr(1);
2 let greeting = "Hello, " + name;
3 /*
4    Application code
5 */
6 div.innerHTML = greeting;
```

Sink

Client Side XSS: Protection

- Solution: Sanitizer

Client Side XSS: Protection

- Solution: Sanitizer
- Removes “dangerous chars” from input

Client Side XSS: Protection

- Solution: Sanitizer
- Removes “dangerous chars” from input
- Hand-written sanitizers dubbed hand sanitizer

Client Side XSS: Protection

- Solution: Sanitizer
- Removes “dangerous chars” from input
- Hand-written sanitizers dubbed hand sanitizer

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4      ...
5  */
6  greeting = sanitize(greeting); Sanitizer
7  /*
8      ...
9  */
10 div.innerHTML = greeting;
```

Client Side XSS: Protection

- Solution: Sanitizer
- Removes "dangerous chars" from input
- Hand-written sanitizers dubbed hand sanitizer

Source

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4      ...
5  */
6  greeting = sanitize(greeting);
7  /*
8      ...
9  */
10 div.innerHTML = greeting;
```

Sanitizer

Sink

Client Side XSS: Protection

- Solution: Sanitizer
- Removes "dangerous chars" from input
- Hand-written sanitizers dubbed hand sanitizer

Source

```
1  let name = location.hash.substr(1);
2  let greeting = "Hello, " + name;
3  /*
4      ...
5  */
6  greeting = sanitize(greeting); Sanitizer
7  /*
8      ...
9  */
10 div.innerHTML = greeting;
```

Sink

Sanitizing: Difficulties

What about this?

Sanitizing: Difficulties

What about this?

```
1  function sanitize(s) {
2    return s.replace("<", "").replace(">", "");
3  }
```

Figure: HTML Sanitizer

Sanitizing: Difficulties

What about this?

```
1 function sanitize(s) {
2   return s.replace("<", "").replace(">", "");
3 }
```

Figure: HTML Sanitizer

Visiting:

foo.com#<img src=x onerror=alert('xss')>

Hello, img src=x onerror=alert('xss')

Sanitizing: Difficulties

What about this?

```
1  function sanitize(s) {
2    return s.replace("<", "").replace(">", "");
3  }
```

Figure: HTML Sanitizer

Visiting:

`foo.com#<><img src=x onerror=alert('xss')>`

Sanitizing: Difficulties

What about this?

```
1  function sanitize(s) {
2    return s.replace("<", "").replace(">", "");
3  }
```

Figure: HTML Sanitizer

Visiting:
`foo.com#<><img src=x onerror=alert('xss')>`

How to sanitize?

- We have 3 injection contexts
  - HTML, HTML attribute and JavaScript

How to sanitize?

- We have 3 injection contexts
- Exploits require different characters per context

How to sanitize?

- We have 3 injection contexts
- Exploits require different characters per context

Characters to be encoded per injection context

<table>
<tr><th>Context</th><th>OWASP Recommendations</th></tr>
<tr><td>HTML</td><td><>'"& except HTML encoded chars</td></tr>
<tr><td>HTML Attr.</td><td>The quote characters (" and ') as well as characters usable to break out of unquoted attribute values (including: [space] % * + , - / ; < = > ^ and |), properties and event handlers</td></tr>
<tr><td>JavaScript</td><td>non-alphanumeric except ,._ whitespace or hex/unicode encoded</td></tr>
</table>

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

⇒ None do what's required...

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

⇒ None do what's required...

- They all encode a subset of "dangerous" characters

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

$\Rightarrow$ None do what's required...

- They all encode a subset of “dangerous” characters

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

$\Rightarrow$ None do what's required...

- They all encode a subset of "dangerous" characters

What now?

Challenges for Developers

The JavaScript standard library has 3 functions that look somewhat related:

- `escape`, `encodeURI`, `encodeURIcomponent`

⇒ None do what's required…

- They all encode a subset of "dangerous" characters

What now?

Hey, folks know regex!

Regular Expressions to the Rescue!

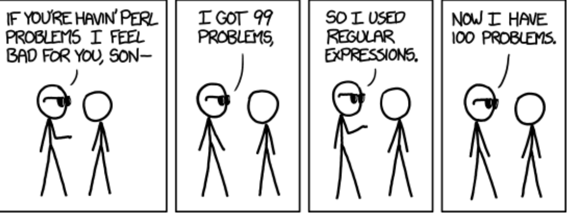


Figure: https://xkcd.com/1171/

Regular Expressions to the Rescue!

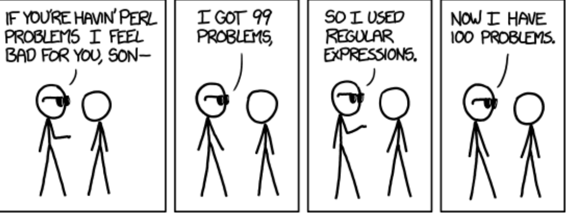


Figure: `https://xkcd.com/1171/`

HTML is not a regular language...

Regular Expressions to the Rescue!

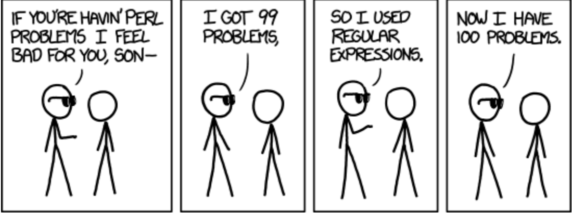


Figure: `https://xkcd.com/1171/`

HTML is not a regular language...

⇒ Regular Expressions unsuited to parse it

Regular Expressions to the Rescue!

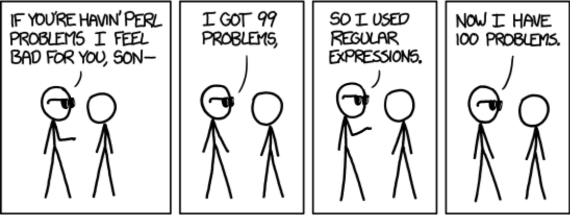


Figure: `https://xkcd.com/1171/`

HTML is not a regular language...

⇒ Regular Expressions unsuited to parse it

- Parsing it requires to build and manipulate a DOM while traversing the input

Sanitizing: Difficulties

Very difficult to get right...

Sanitizing: Difficulties

Very difficult to get right. . .

> *More than half of the DOM XSS root causes were due to bugs in HTML sanitizers*
>
> —*Google Research: Trusted Types - mid 2021 report*

Large Scale Study

State of Sanitization on the Web

We asked ourselves two questions:

State of Sanitization on the Web

We asked ourselves two questions:

Q1: How prevalent are sanitizers?

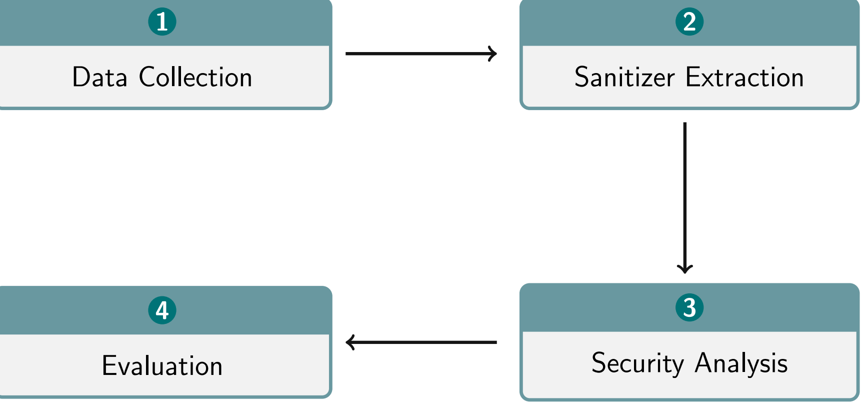
State of Sanitization on the Web

We asked ourselves two questions:

Q1: How prevalent are sanitizers?

Q2: Are they actually secure?

Study: Setup



Project Foxhound

- Firefox fork enhanced with taint-tracking capabilities

Project Foxhound

- Firefox fork enhanced with taint-tracking capabilities
- Also records all operations that occurred on tainted values
  - Deep insight into inner working of web application

Project Foxhound

- Firefox fork enhanced with taint-tracking capabilities
- Also records all operations that occurred on tainted values
  - Deep insight into inner working of web application
- Open source, actively maintained and compatible with Playwright

  $\Rightarrow$ good addition to security testing toolbelt

1 Data Collection

- Take our taint browser

1 Data Collection

- Take our taint browser
- Visit top 20 000 websites

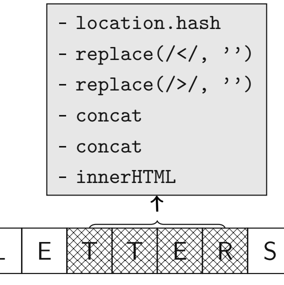
1 Data Collection

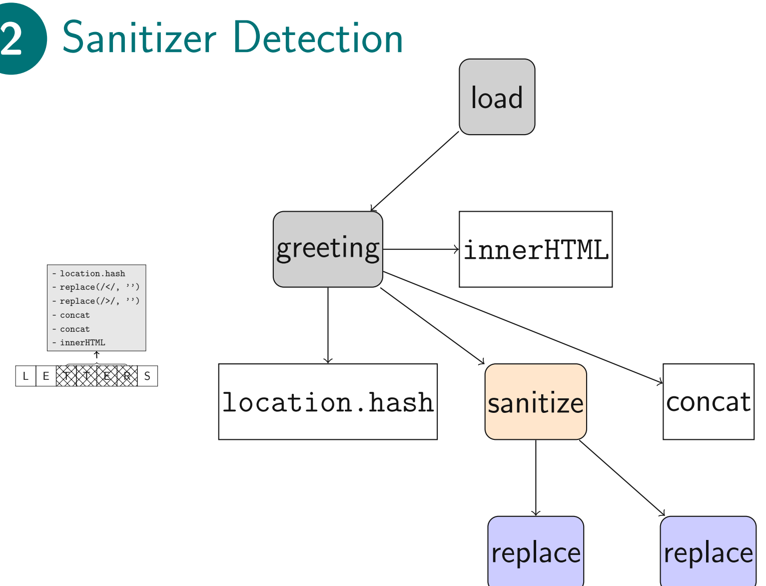
- Take our taint browser
- Visit top 20 000 websites
- Record data flows relevant to Client-Side XSS

1 Data Collection

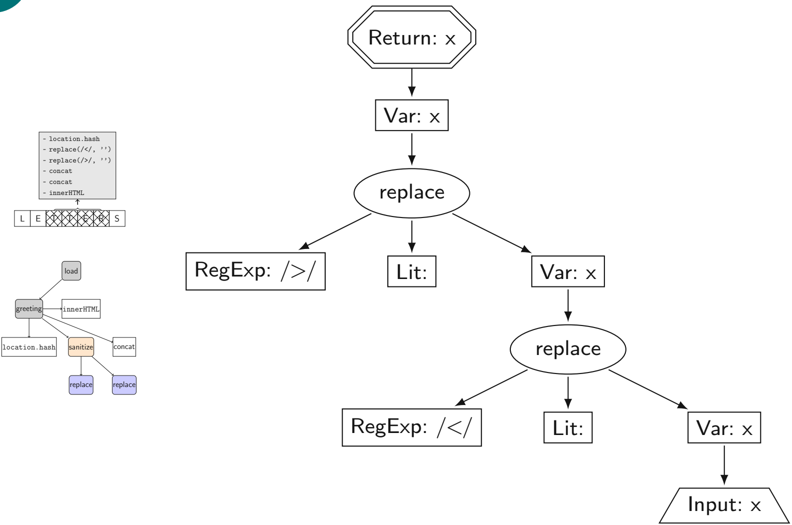
- Take our taint browser
- Visit top 20 000 websites
- Record data flows relevant to Client-Side XSS
  - Occurred on 3887 domains

2 Sanitizer Detection

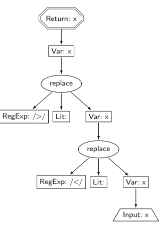



2
Sanitizer Detection
load
greeting
innerHTML
location.hash
sanitize
concat
replace
replace
- location.hash
- replace(/</, '')
- replace(/>/, '')
- concat
- concat
- innerHTML
L E T T E R S

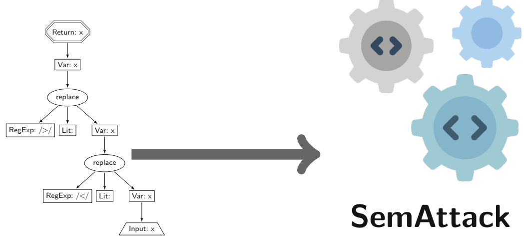
2 Sanitizer Detection



3 Sanitizer Analysis



3 Sanitizer Analysis



3 Sanitizer Analysis

Return: x

Var: x

replace

RegExp: />/ Lit: Var: x

replace

RegExp: /</ Lit: Var: x

Input: x

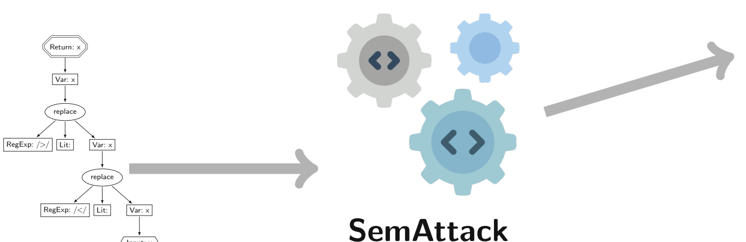
SemAttack

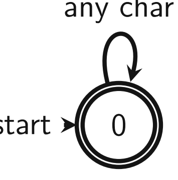
any char

start 0

Post Image

3 Sanitizer Analysis





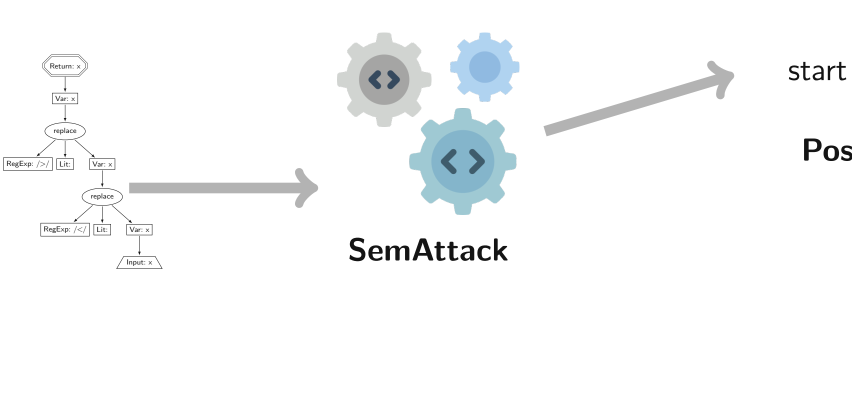
Post Image

`<img src=x onerror=alert('XSS')>`

XSS Payload

3 Sanitizer Analysis

any char



start

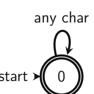
0

Post Image

<img src=x onerror=alert('XSS')>

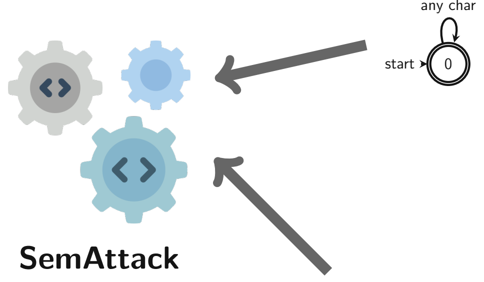
XSS Payload

4 Validation

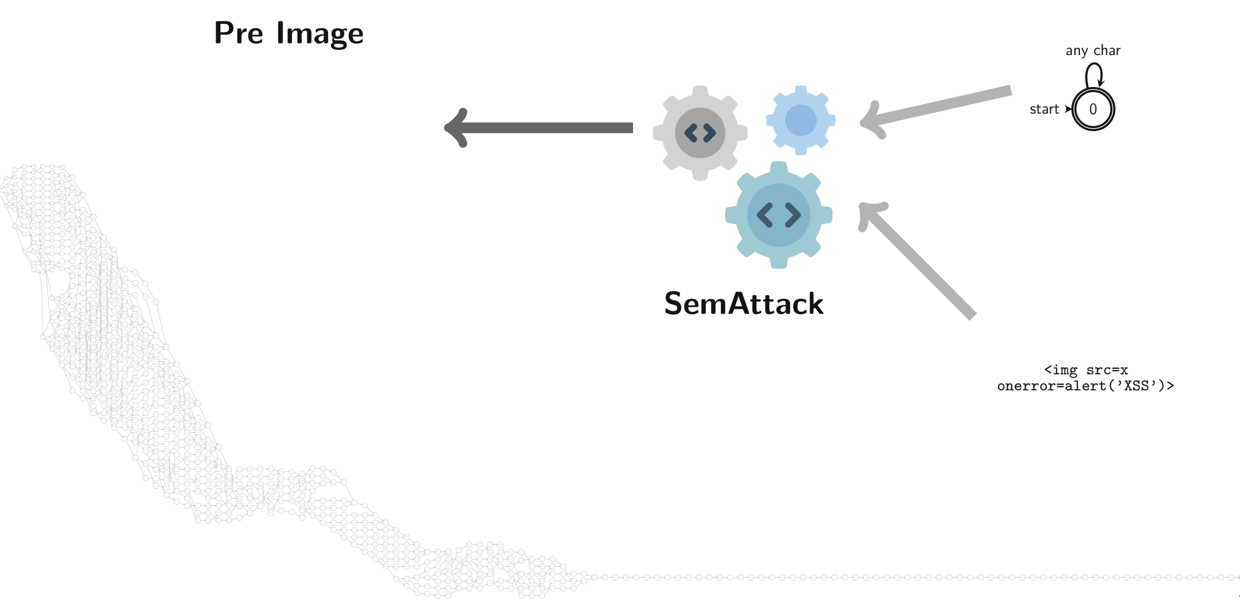



```
<img src=x
onerror=alert('XSS')>
```

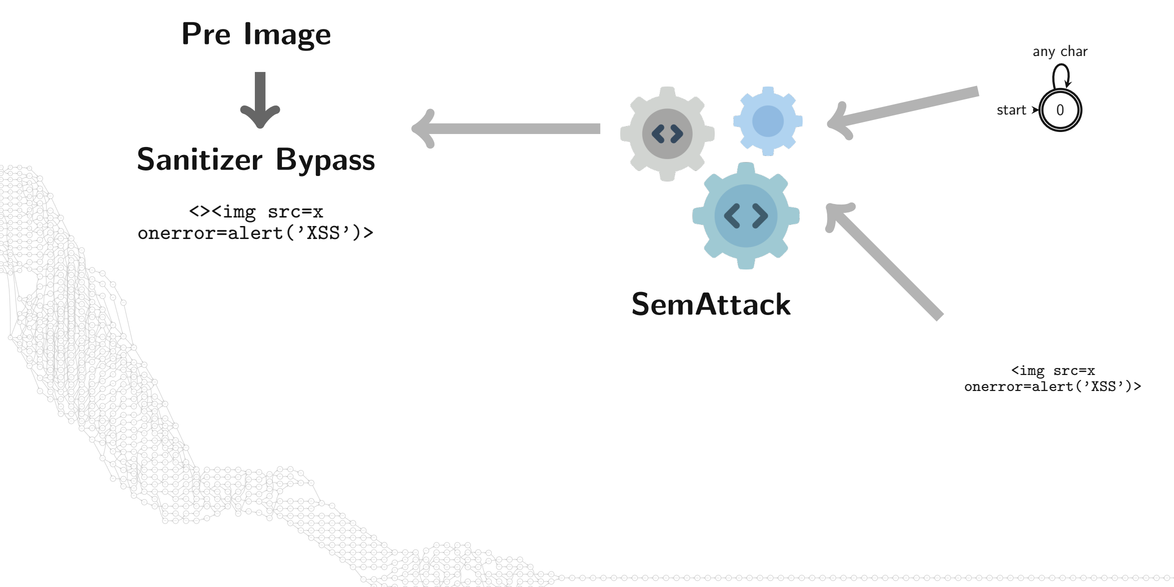
4 Validation



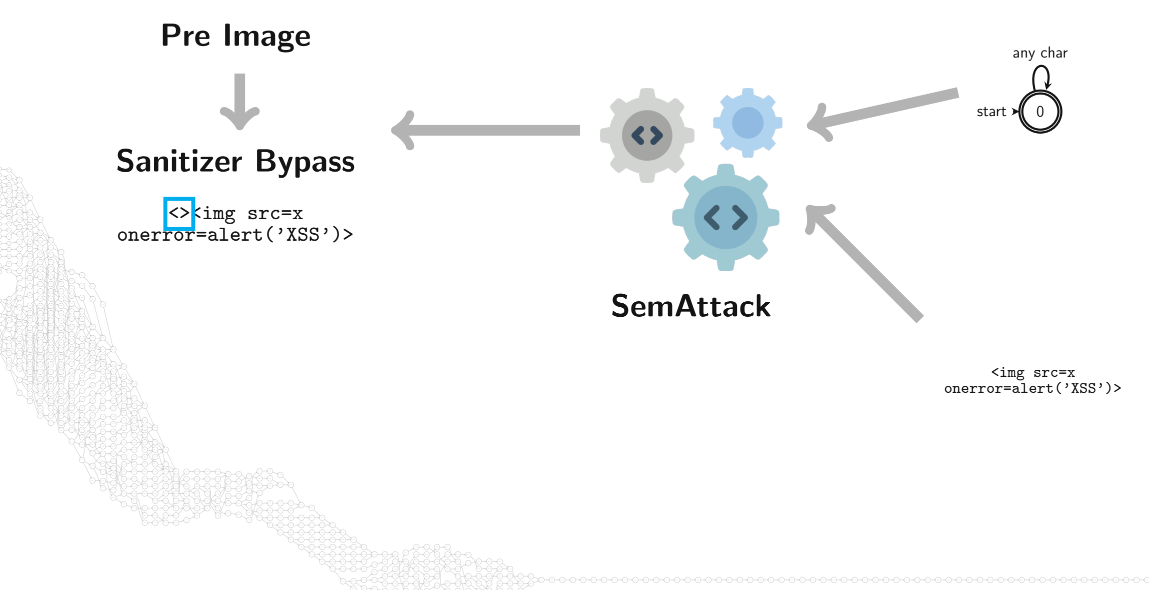
4 Validation



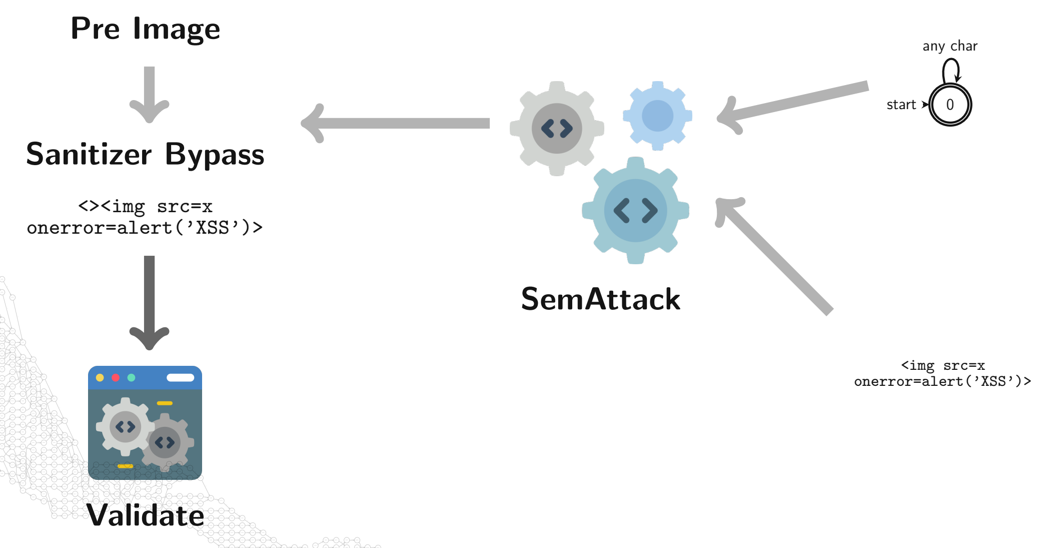
4 Validation



4 Validation



4 Validation



Results

- 3887 out of 20 000 websites contained interesting data flows.

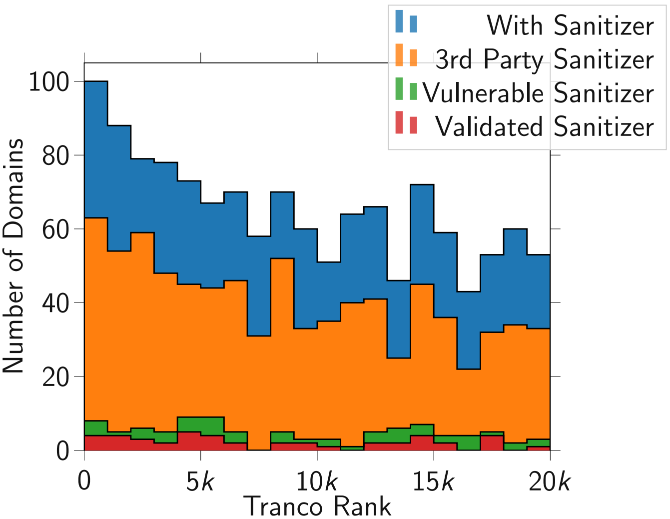
Results

- 3887 out of 20 000 websites contained interesting data flows.
- We found 705 unique sanitizers on 1415 out of those 3887 domains.

Results

- 3887 out of 20 000 websites contained interesting data flows.
- We found 705 unique sanitizers on 1415 out of those 3887 domains.
- 88 sanitizers on 102 domains detected as insecure by SemAttack.

Results



Cabinet of Horrors

Effective against germs, but not against XSS payloads!

Optimized for specific Payload

```
1 function f(v) {
2   return v.replace(/'/g, "").replace(/\(/g, "")
3   .replace(/\)/g, "").replace(/alert/g, "");
4 }
```

Optimized for specific Payload

Delete all Single Quotes

```
1  function f(v) {
2    return v.replace(/'/g, "").replace(/\(/g, "")
3    .replace(/\)/g, "").replace(/alert/g, "");
4  }
```

Optimized for specific Payload

```
1 function f(v) {
2   return v.replace(/'/g, "").replace(/\(/g, "")
3   .replace(/\)/g, "").replace(/alert/g, "");
4 }
```

Delete all Parentheses

Optimized for specific Payload

```
1   function f(v) {
2     return v.replace(/'/g, "").replace(/\(/g, "")
3     .replace(/\)/g, "").replace(/alert/g, "");
4   }
```

Delete dangerous payload

Optimized for specific Payload

```
1 function f(v) {
2   return v.replace(/'/g, "").replace(/\(/g, "")
3   .replace(/\)/g, "").replace(/alert/g, "");
4 }
```

Issues:

- Real hackers do not use alert

Optimized for specific Payload

```
1  function f(v) {
2    return v.replace(/'/g, "").replace(/\(/g, "")
3    .replace(/\)/g, "").replace(/alert/g, "");
4  }
```

Issues:

- Real hackers do not use alert
- Removing Parentheses to prevent function calls seems reasonable?

Optimized for specific Payload

```
1  function f(v) {
2    return v.replace(/'/g, "").replace(/\(/g, "")
3    .replace(/\)/g, "").replace(/alert/g, "");
4  }
```

Issues:

- Real hackers do not use alert
- Removing Parentheses to prevent function calls seems reasonable?
  - But. . . JavaScript is crazy

Optimized for specific Payload

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

Issues:

- Real hackers do not use alert
- Removing Parentheses to prevent function calls seems reasonable?
  - But... JavaScript is crazy
  - `` confirm`xss` `` works as well

Wrong Context

```
function sanitize(v) {
  return v.replace(/</g, "&lt;")
    .replace(/>/g, "&gt;");
}
var url = 'http://example.org;cat=' +
    sanitize(cat) + '?';
document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Wrong Context

```
function sanitize(v) {
  return v.replace(/</g, "&lt;")
    .replace(/>/g, "&gt;");
}
var url = 'http://example.org;cat=' +
    sanitize(cat) + '?';
document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Encode all angle brackets

Issues:

- Encoding angle brackets generally a good idea!

Wrong Context

```
function sanitize(v) {
  return v.replace(/</g, "&lt;")
    .replace(/>/g, "&gt;");
}
var url = 'http://example.org;cat=' +
    sanitize(cat) + '?';
document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Injection Context

Issues:

- Encoding angle brackets generally a good idea!
  - But... context is inside an attribute

Wrong Context

```
1  function sanitize(v) {
2    return v.replace(/</g, "&lt;")
3      .replace(/>/g, "&gt;");
4  }
5  var url = 'http://example.org;cat=' +
6      sanitize(cat) + '?';
7  document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Issues:

- Encoding angle brackets generally a good idea!
  - But. . . context is inside an attribute
  - No angle brackets required to break out and inject payload

Wrong Context

```
1  function sanitize(v) {
2    return v.replace(/</g, "&lt;")
3      .replace(/>/g, "&gt;");
4  }
5  var url = 'http://example.org;cat=' +
6      sanitize(cat) + '?';
7  document.write('<iframe src="' + url + '" style="display:none"></iframe>');
```

Issues:

- Encoding angle brackets generally a good idea!
  - But... context is inside an attribute
  - No angle brackets required to break out and inject payload
  - Example: `" onload=alert(1) foo=`

Blocklisting

```
v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
  .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
  .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

Blocklisting

Delete all `img` tags with content

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

Blocklisting

Delete all a tags with content

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

Blocklisting

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Delete all `script` tags with content

Issues:

Blocklisting

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

- Blocklisting is brittle by nature

Blocklisting

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

- Blocklisting is brittle by nature
- Several other tags can be used to inject payloads:

Blocklisting

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Issues:

- Blocklisting is brittle by nature
- Several other tags can be used to inject payloads:
  - E.g., `<image>` behaves exactly the same as `<img>`

Blocklisting

```
1  v = decodeURIComponent(location.hash.replace('#', '').split('/')[2]);
2  v = v.replace(/<img(.*)?(\/)?>(.*)?(<\/img>)?/gi, '')
3    .replace(/<a(.*)?(\/)?>(.*)?(<\/a>)?/gi, '')
4    .replace(/<script(.*)?(\/)?>(.*)? (<\/script>)?/gi, '');
```

Matching closing tags

Issues:

- Blocklisting is brittle by nature
- Several other tags can be used to inject payloads:
  - E.g., `<image>` behaves exactly the same as `<img>`

Small aside:

- HTML parsers accept attributes in end tags (and ignore them)

Regular Expressions Limitations

```
1  var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2    .replace('"', "")
3    .replace(">", "")
4    .replace("#", "")
5    .replace("<", "");
6  document.write('<script type="text/javascript" src="example.org?url='+url+'"
   ↪  ></script>');
```

Issues:

Regular Expressions Limitations

Delete all `script`, `a` and `img` tags

```
1 var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2   .replace('"', "")
3   .replace(">", "")
4   .replace("#", "")
5   .replace("<", "");
6 document.write('<script type="text/javascript" src="example.org?url='+url+'"
  ↪  ></script>');
```

Issues:

Regular Expressions Limitations

```
1   var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2     .replace('"', "")
3     .replace(">", "")
4     .replace("#", "")
5     .replace("<", "");
6   document.write('<script type="text/javascript" src="example.org?url='+url+'"
    ↪  ></script>');
```

Issues:

- Regular Expressions do a single scan over the input

Regular Expressions Limitations

```
1  var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2    .replace('"', "")
3    .replace(">", "")
4    .replace("#", "")
5    .replace("<", "");
6  document.write('<script type="text/javascript" src="example.org?url='+url+'"
   ↪  ></script>');
```

Issues:

- Regular Expressions do a single scan over the input
  - E.g., `<sc<a>ript>` would only have the inner tag removed

Regular Expressions Limitations

```
1  var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2    .replace('"', "")
3    .replace(">", "")
4    .replace("#", "")
5    .replace("<", "");
6  document.write('<script type="text/javascript" src="example.org?url='+url+'"
   ↪  ></script>');
```

Remove dangerous characters

Issues:

- Regular Expressions do a single scan over the input
- The JavaScript API for `replace()` is somewhat unintuitive

Regular Expressions Limitations

```
1   var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2     .replace('"', "")
3     .replace(">", "")
4     .replace("#", "")
5     .replace("<", "");
6   document.write('<script type="text/javascript" src="example.org?url='+url+'"
↪   ></script>');
```

Issues:

- Regular Expressions do a single scan over the input
- The JavaScript API for `replace()` is somewhat unintuitive
  - `replace("<", "")` replaces only the first occurrence of <

Regular Expressions Limitations

```
var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
  .replace('"', "")
  .replace(">", "")
  .replace("#", "")
  .replace("<", "");
document.write('<script type="text/javascript" src="example.org?url='+url+'"
↪ ></script>');
```

Issues:

- Regular Expressions do a single scan over the input
- The JavaScript API for `replace()` is somewhat unintuitive
  - `replace("<", "")` replaces only the first occurrence of <
  - To replace all, `replace(/</g, "")` has to be used

Regular Expressions Limitations

```
1 var url = location.href.replace(/<script[\S\s]*?\1>|<\/?(a|img)[^>]*>/gi, "")
2   .replace('"', "")
3   .replace(">", "")
4   .replace("#", "")
5   .replace("<", "");
6 document.write('<script type="text/javascript" src="example.org?url='+url+'"
  ↪  ></script>');
```

Issues:

- Regular Expressions do a single scan over the input
- The JavaScript API for `replace()` is somewhat unintuitive
  - `replace("<", "")` replaces only the first occurrence of <
  - To replace all, `replace(/</g, "")` has to be used
  - One of the most frequent errors we encountered!

Mitigations

HTML Parser to Sanitize

I'll just use a HTML parser library to sanitize my input!

HTML Parser to Sanitize

I'll just use a HTML parser library to sanitize my input!

HTML Parser to Sanitize

I'll just use a HTML parser library to sanitize my input!

- Does this HTML parser actually behave like your visitor's browsers do?

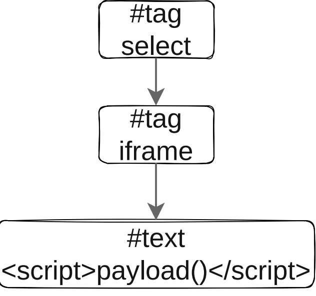
Parser Confusion to Sanitizer Bypass

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

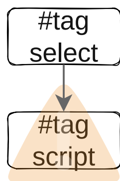
Parser Confusion to Sanitizer Bypass

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

Parsed by Google Caja



Parsed by Google Chrome



Parser Confusion to Sanitizer Bypass: 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.

Parser Confusion to Sanitizer Bypass: Root Cause

The ”nothing” content model

When an element's content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

Parser Confusion to Sanitizer Bypass: Root Cause

The ”nothing” content model
When an element's content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

However. . .

Parser Confusion to Sanitizer Bypass: Root Cause

The "nothing" content model
When an element's content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

However... the parsing specification says content of `iframe` should be parsed as text...

Parser Confusion to Sanitizer Bypass: Root Cause

The "nothing" content model
When an element's content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

However... the parsing specification says content of `iframe` should be parsed as text...

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

: no code execution

Parser Confusion to Sanitizer Bypass: Root Cause

The "nothing" content model

When an element's content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

However... the parsing specification says content of `iframe` should be parsed as text...

`` div.innerHTML = `<iframe><img src=x onerror=alert(1)>`; `` : no code execution

So the sanitizer is actually correct, but...

Parser Confusion to Sanitizer Bypass: Root Cause

The ”nothing” content model
When an element’s content model is nothing, the element must contain no Text nodes (other than inter-element whitespace) and no element nodes.

However. . . the parsing specification says content of `iframe` should be parsed as text. . .

`` div.innerHTML = `<iframe><img src=x onerror=alert(1)>`; `` : no code execution

So the sanitizer is actually correct, but. . .

Where has the `iframe` gone actually?

The missing `iframe`

Recall the payload: `<select><iframe><script>payload()`

The missing `iframe`

Recall the payload: `<select><iframe><script>payload()`

the `select` element

Content model:

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

The missing `iframe`

Recall the payload: `<select><iframe><script>payload()`

the `select` element

Content model:

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

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

The missing `iframe`

Recall the payload: `<select><iframe><script>payload()`

> the `select` element
>
> **Content model:**
>
> *Zero or more `option`, `optgroup`, and script-supporting elements.*

“script-supporting elements” are `script` and `template` tags

Thus, an `iframe` can't be a child of `select`

The missing `iframe`

Recall the payload: `<select><iframe><script>payload()`

> the `select` element
>
> **Content model:**
>
> *Zero or more `option`, `optgroup`, and script-supporting elements.*

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

Thus, an `iframe` can't be a child of `select`, and Chrome drops it

Parser Confusion to Sanitizer Bypass: Summary

Sanitization based on a “full” HTML parser needs to take into account:

- All subtleties of the HTML specification
  - It’s a 1300+ page document...
- How browsers diverge from it

Parser Confusion to Sanitizer Bypass: Summary

Sanitization based on a “full” HTML parser needs to take into account:

- All subtleties of the HTML specification
  - It’s a 1300+ page document...
- How browsers diverge from it

⇒ This also applies to server-side HTML sanitization!

How to protect yourself then?

Protection

1. Avoid the need to sanitize!

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection
   - Consider e.g., markdown for formatted input

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection
   - Consider e.g., markdown for formatted input
   - This allows to “just” encode everything

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection
   - Consider e.g., markdown for formatted input
   - This allows to “just” encode everything
   - Please recall the DOM clobbering talk!

Protection

1. Avoid the need to sanitize!
   – Minimize putting user data into the DOM
2. Avoid HTML markup injection
   – Consider e.g., markdown for formatted input
   – This allows to "just" encode everything
   – Please recall the DOM clobbering talk!
3. Use a well tested library

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection
   - Consider e.g., markdown for formatted input
   - This allows to "just" encode everything
   - Please recall the DOM clobbering talk!
3. Use a well tested library
   - E.g., DOMPurify

Protection

1. Avoid the need to sanitize!
   - Minimize putting user data into the DOM
2. Avoid HTML markup injection
   - Consider e.g., markdown for formatted input
   - This allows to “just” encode everything
   - Please recall the DOM clobbering talk!
3. Use a well tested library
   - E.g., DOMPurify
   - Keep it up to date!

Way Forward?

Two upcoming browser features:

- Sanitizer API
- Trusted Types

Sanitizer API

- Buy in by Mozilla, Google and Microsoft. Safari has not implemented it (yet?)

Sanitizer API

- Buy in by Mozilla, Google and Microsoft. Safari has not implemented it (yet?)

Goal: Adding a robust and secure by default HTML sanitizer

Sanitizer API

- Buy in by Mozilla, Google and Microsoft. Safari has not implemented it (yet?)

Goal: Adding a robust and secure by default HTML sanitizer

- Updated with the browser, so any bypasses are fixed automatically

Sanitizer API: Usage

```
1  let node = document.createElement('div');
2  let sanitizer = new Sanitizer(); Create Sanitizer object
3  let payload = '<img src=x onerror=alert(1)>';
4  node.setHTML(payload, sanitizer);
5  let sanitized = sanitizer.sanitizeFor('div', payload);
6  node.replaceChildren(...sanitized.childNodes);
7  // innerHTML of node is: <img src="x"> after both calls
```

Figure: Usage of the Sanitizer API

Sanitizer API: Usage

```
1  let node = document.createElement('div');
2  let sanitizer = new Sanitizer();
3  let payload = '<img src=x onerror=alert(1)>';
4  node.setHTML(payload, sanitizer); Option 1
5  let sanitized = sanitizer.sanitizeFor('div', payload);
6  node.replaceChildren(...sanitized.childNodes);
7  // innerHTML of node is: <img src="x"> after both calls
```

Figure: Usage of the Sanitizer API

Sanitizer API: Usage

```
1 let node = document.createElement('div');
2 let sanitizer = new Sanitizer();
3 let payload = '<img src=x onerror=alert(1)>';
4 node.setHTML(payload, sanitizer);
5 let sanitized = sanitizer.sanitizeFor('div', payload);
6 node.replaceChildren(...sanitized.childNodes);
7 // innerHTML of node is: <img src="x"> after both calls
```

Option 2

Figure: Usage of the Sanitizer API

Sanitizer API: Takeaways

Positives:

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

- `string` to `string` sanitization not supported

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

- `string` to `string` sanitization not supported
  - ⇒ Does not match how a lot of applications are written!

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

$\rightarrow$ It uses exactly the same HTML parser as the browser

$\rightarrow$ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

- `string` to `string` sanitization not supported
  - $\Rightarrow$ Does not match how a lot of applications are written!

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

- `string` to `string` sanitization not supported
  - ⇒ Does not match how a lot of applications are written!

Still under development

Sanitizer API: Takeaways

Positives:

Secure by default & no footgun potential

→ It uses exactly the same HTML parser as the browser

→ So not divergences in parsing behavior possible

Negatives:

Enforces secure usage via API

- `string` to `string` sanitization not supported
  - ⇒ Does not match how a lot of applications are written!

Still under development

- Not yet ready for productive use

Trusted Types

Trusted Types

Idea: Ensure sanitization via strong typing

Trusted Types

Idea: Ensure sanitization via strong typing

Buy in only by Google, Microsoft

Trusted Types

Idea: Ensure sanitization via strong typing

Buy in only by Google, Microsoft

- Make sinks accept Trusted values instead of strings
- Assigning strings gives a type error

Trusted Types: Enforcement

Changing API for `.innerHTML` hugely invasive!

Trusted Types: Enforcement

Changing API for `.innerHTML` hugely invasive!

- Breaks legacy code

Trusted Types: Enforcement

Changing API for `.innerHTML` hugely invasive!

- Breaks legacy code
- Solution: Allow website to opt-into enforcement

Trusted Types: Enforcement

Changing API for `.innerHTML` hugely invasive!

- Breaks legacy code
- Solution: Allow website to opt-into enforcement

Add a Content Security Policy (CSP) directive!

Trusted Types: Enforcement

Changing API for `.innerHTML` hugely invasive!

- Breaks legacy code
- Solution: Allow website to opt-into enforcement

Add a Content Security Policy (CSP) directive!

```
require-trusted-types-for 'script';
```

Trusted Types: Usage

```
1  const p = '<img src=x onerror=alert(1)>';
2  htmlPolicy = trustedTypes.createPolicy('sanitize', {
3      createHTML: s => s.replace(/\</g, '&lt;')
4  });
5  node.innerHTML = htmlPolicy.createHTML(p);
6  node.innerHTML = p;
```

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Usage

```
1 const p = '<img src=x onerror=alert(1)>';
2 htmlPolicy = trustedTypes.createPolicy('sanitize', {
3     createHTML: s => s.replace(/\</g, '&lt;')
4 });
5 node.innerHTML = htmlPolicy.createHTML(p);
6 node.innerHTML = p;
```

Create a Trusted Types Policy

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Usage

```
1  const p = '<img src=x onerror=alert(1)>';
2  htmlPolicy = trustedTypes.createPolicy('sanitize', {
3      createHTML: s => s.replace(/\</g, '&lt;')
4  });
5  node.innerHTML = htmlPolicy.createHTML(p);
6  node.innerHTML = p;
```

Define a Sanitizer for HTML sinks

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Usage

```
1  const p = '<img src=x onerror=alert(1)>';
2  htmlPolicy = trustedTypes.createPolicy('sanitize', {
3      createHTML: s => s.replace(/\</g, '&lt;')
4  });
5  node.innerHTML = htmlPolicy.createHTML(p);
6  node.innerHTML = p;
```

Use Policy to create TrustedHTML

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Usage

```
1  const p = '<img src=x onerror=alert(1)>';
2  htmlPolicy = trustedTypes.createPolicy('sanitize', {
3      createHTML: s => s.replace(/\</g, '&lt;')
4  });
5  node.innerHTML = htmlPolicy.createHTML(p);
6  node.innerHTML = p;
```

Secure Assignment

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Usage

```
1  const p = '<img src=x onerror=alert(1)>';
2  htmlPolicy = trustedTypes.createPolicy('sanitize', {
3      createHTML: s => s.replace(/\</g, '&lt;')
4  });
5  node.innerHTML = htmlPolicy.createHTML(p);
6  node.innerHTML = p;  Insecure Assignment
```

Figure: Creating and Using a Trusted Types Policy

Trusted Types: Takeaways

Enforcing sanitization fantastic idea!

Trusted Types: Takeaways

Enforcing sanitization fantastic idea!

- No suggestions about sanitization itself
- The broken ones shown before could be used as policy $\rightarrow$ false sense of security
- Requires a “generic” sanitizer
- Usability Issues
  - Do you know about the undocumented parameters?

Trusted Types: Takeaways

Enforcing sanitization fantastic idea!

- No suggestions about sanitization itself
- The broken ones shown before could be used as policy $\rightarrow$ false sense of security
- Requires a “generic” sanitizer
- Usability Issues
  - Do you know about the undocumented parameters?

Google and Microsoft only technology

Trusted Types: Takeaways

Enforcing sanitization fantastic idea!

- No suggestions about sanitization itself
- The broken ones shown before could be used as policy → false sense of security
- Requires a "generic" sanitizer
- Usability Issues
  - Do you know about the undocumented parameters?

Google and Microsoft only technology

→ Idea is you get Trusted Types for free when using frameworks such as Angular

Key Takeaways

- Client-Side XSS still an issue

Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal

Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones

Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones
- Developers misunderstand key aspects of JavaScript, including:

Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones
- Developers misunderstand key aspects of JavaScript, including:
  - URL encoding functionality: `escape`, `encodeURI(Component)`

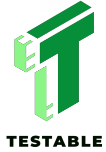
Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones
- Developers misunderstand key aspects of JavaScript, including:
  - URL encoding functionality: `escape`, `encodeURI(Component)`
  - Regular expression usage

Key Takeaways

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones
- Developers misunderstand key aspects of JavaScript, including:
  - URL encoding functionality: `escape`, `encodeURI(Component)`
  - Regular expression usage
  - Parts of the standard library

Thank you for your attention!



Resources

github.com/SAP/project-foxhound
github.com/ias-tubs/hand_sanitizer

Contact

david.klein@tu-braunschweig.de
david-klein-b2aa80254
twitter.com/ncd_leen

Summary

- 3887 out of 20 000 websites contained interesting data flows.
- We found 705 unique sanitizers on 1415 out of those 3887 domains.
- 88 sanitizers on 102 domains detected as insecure by SemAttack.

- Client-Side XSS still an issue
- Deployed sanitizers are neither generic nor minimal
- First party sanitizers more likely to be vulnerable than third party ones
- Developers misunderstand key aspects of JavaScript