How CSP Will (maybe) Solve the XSS Problem

Neil Matatall
@owaspoc 2/20/2014

XSS is so 1998

So is XSS prevention

What is XSS

```
<input name="id"
value="${id}">
```

What is XSS

”><script>badStuff()</script>

What is XSS

<input name="id" value="">

<script>badStuff()</script>

">

What is XSS

```
<script>
  var id=${id};
</script>
```

What is XSS

0;badStuff();

What is XSS

<script>
  var id=0; badStuff()
</script>

What is XSS

```
<script>
  setInterval(1000, ‘${todo}’)
</script>
```

What is XSS

badStuff();

What is XSS

<script>

setInterval(1000, '**badStuff()**')

</script>

What is XSS

```
<input name="id"
  onBlur="doStuff(${id})">
```

What is XSS

); badStuff(

What is XSS

```
<input name="id"
    onBlur="doStuff(); badStuff()">
```

What is XSS

Hi Mom!

What is XSS

javascript:badThings()

What is XSS

Hi Mom!

What is XSS

```
<a href="#"
onClick="doThing('${link}')">
   Hi Mom!
</a>
```

What is XSS

');badStuff('

What is XSS

```
<a href="#" onClick=
"doThing('&#39;);badStuff(&#39;')">
  Hi Mom!
</a>
```

What is XSS

```
<a href="#" onClick=
"doThing(");badStuff(")">
   Hi Mom!
</a>
```

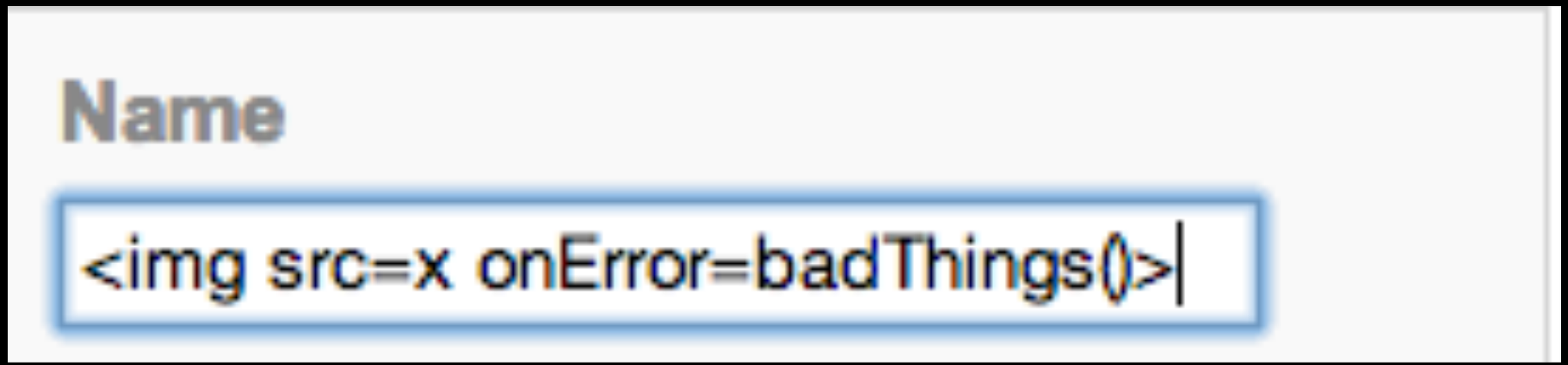
What is DOM XSS

```
<script>

document.body.innerHTML =
 document.getElementById('name').value;

</script>
```

What is DOM XSS



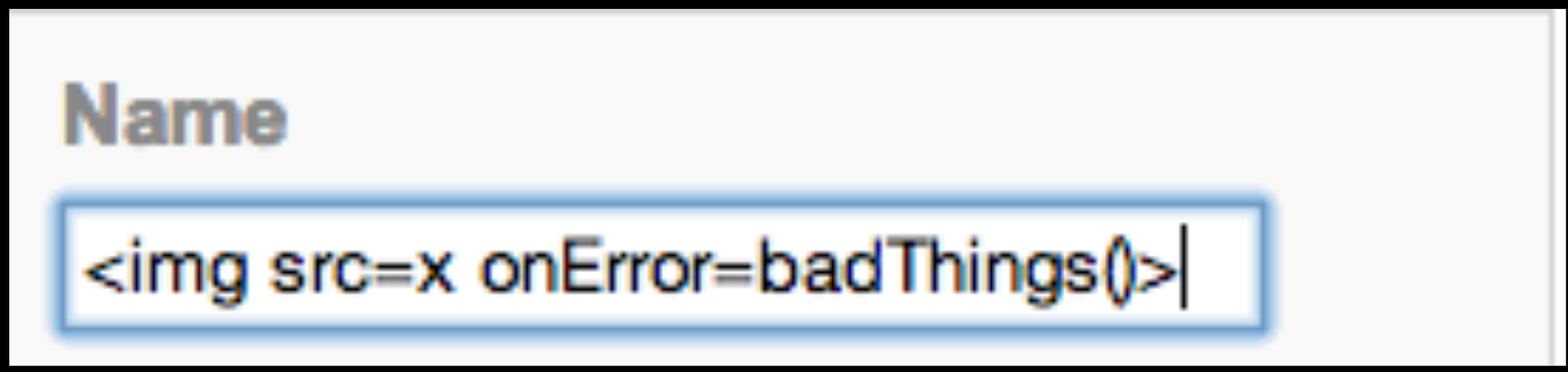
What is DOM XSS

<body>
<img src=x onError=badThings()>
</body>

What is DOM XSS

```
<script>
  var name = $('#name').val
  $('body').html(name)
</script>
```

What is DOM XSS



What is DOM XSS

<body>
<img src=x onError=badThings()>
</body>

jQuery is XSS

$(data)

What is XSS

```
<style>

a {
  color: ${usersFavoriteColor};
}
</style>
```

What is XSS

I dunno, something with SVG, CSS Expressions, etc. The list grows.

Because Dr. Mario

Scriptless Attacks – Stealing the Pie Without Touching the Sill

Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, Jörg Schwenk
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Due to their high practical impact, Cross-Site Scripting (XSS) attacks have attracted a lot of attention from the security community members. In the same way, a plethora of more or less effective defense techniques have been proposed, addressing the causes and effects of XSS vulnerabilities.

As a result, an adversary often can no longer inject or even execute arbitrary scripting code in several real-life scenarios.

In this paper, we examine the attack surface that remains after XSS and similar scripting attacks are supposedly mitigated by preventing an attacker from executing JavaScript code. We address the question of whether an attacker really needs JavaScript or similar functionality to perform attacks aiming for information theft. The surprising result is that an attacker can also abuse Cascading Style Sheets (CSS) in combination with other Web techniques like plain HTML, inactive SVG images or font files. Through several case

Keywords

Scriptless Attacks, XSS, CSS, SVG, HTML5, Attack Fonts

1. INTRODUCTION

In the era of Web 2.0 technologies and cloud computing, a rich set of powerful online applications is available at our disposal. These Web applications allow activities such as online banking, initiating commercial transactions at the online stores, composing e-mails which may contain sensitive information, or even managing personal medical records online. It is therefore only natural to wonder what kind of measures are necessary to protect such data, especially in connection with security and privacy concerns.

A prominent real-life attack vector is *Cross-Site Scripting* (XSS), a type of injection attack in which an adversary injects malicious scripts into an otherwise benign (and trusted)

Why do those crazy things?

CODE != DATA

“text”

“select * from table where id = “ + id

But we don’t do stupid things

I will religiously escape content

I will religiously escape content part 2

XSS Prevention Rules

- 2.1 RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
- 2.2 RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Co
- 2.3 RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common
- 2.4 RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Dat
  - 2.4.1 RULE #3.1 - HTML escape JSON values in an HTML context and read the
    - 2.4.1.1 JSON entity encoding
    - 2.4.1.2 HTML entity encoding
- 2.5 RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into
- 2.6 RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Paramet
- 2.7 RULE #6 - Sanitize HTML Markup with a Library Designed for the Job
- 2.8 RULE #7 - Prevent DOM-based XSS
- 2.9 Bonus Rule #1: Use HTTPOnly cookie flag
- 2.10 Bonus Rule #2: Implement Content Security Policy

I will religiously escape content part 3

JSFuck – Write any JavaScript with 6 Characters: []()!+

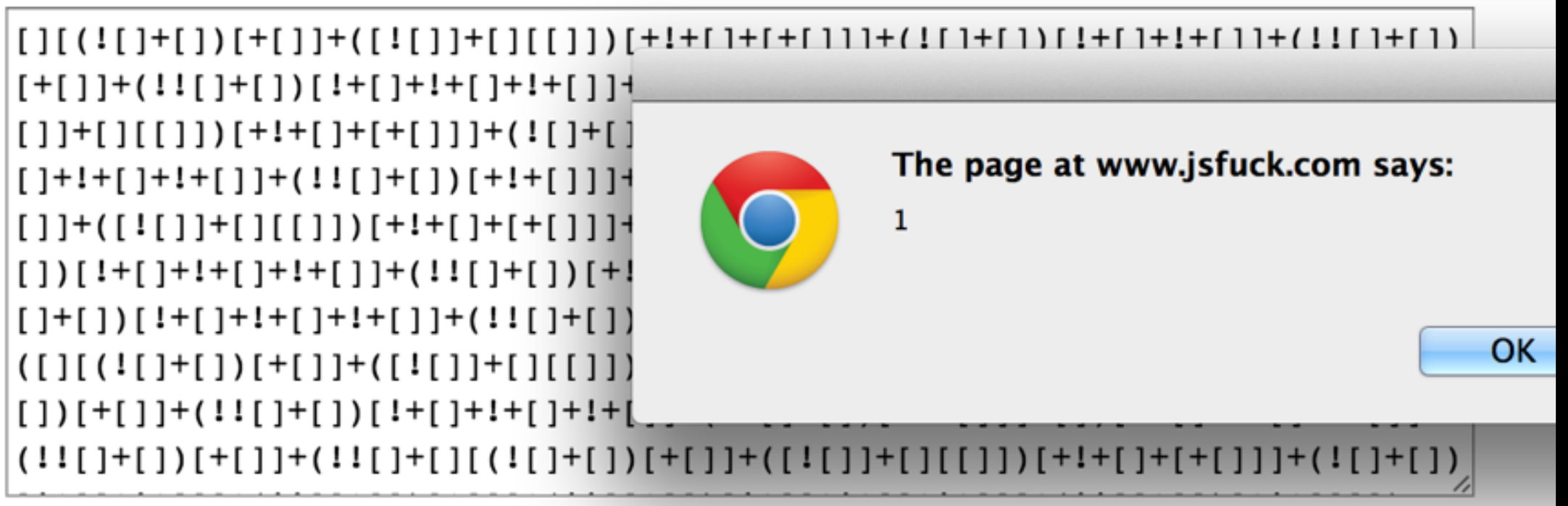
JSFuck

JSFuck is an esoteric and educational programming style based on the atomic parts of JavaScript. It uses only six different characters to write and execute code.

It does not depend on a browser, so you can even run it on Node.js.

Use the form below to convert your own script. Uncheck "eval source" to get back a plain string.

alert(1) Encode Eval Source



1227 chars Run This

I'll sanitize / validate input

I'll use a scanner

I'll perform periodic assessments

Security is about layers

Daynebberaxeded ▸ Melani... 2 ★

Everyone knows that insulating products, like "thermoses" and koozies help keep hot things hot and cold things cold, but the burning question: How do they know?

Thermos? Koozie?

How does the browser know?

<script>goodStuff()</script>
<script>badStuff()</script>

CSP To The Rescue

CSP?

Why CSP?

What is dangerous?

- inline javascript
  - <script>…</script>
  - <input onBlur="…">
  - <a href="javascript:…">
- on-the-fly code generation
  - setTimeout, eval, new Function("…")

“Doctor, it hurts when I do this”
“Don’t do that”

CSP is more than XSS protection

Nothing is free

Removing the dangerous stuff

Report-Only mode

Techniques for removing inline javascript

CSP 1.1

- Whitelisting inline <script> in a safe way

Inline code

```
<script>
  stuff()
</script>
```

Nonces

<script nonce="34298734…">
  stuff()
</script>

Hashes

```
<script>
  stuff()
</script>
```

Hashes are more secure than nonces

What you still can’t do

- Inline event handlers
  - <input onBlur=“doGoodThing()”>
- <a href=“javascript:…”>
- Dynamic javascript
  - <script> var id=${id} </script>
    - Hash values won’t match
    - Nonce provides absolutely no security

Automatic CSP Protection (Silverish bullet)

Whitelisting javascript

- Find all javascript
- Compute all hash values
- Whitelist scripts with corresponding hashes

Assume: Sane web framework

- Do a regular expression search over all templates, capture all inline javascript
- Store a map of the hash(es) in each individual file
- Each time the file is rendered, add the corresponding hashes to the header

Developer productivity

- Serve dynamic hash values in (!production), serve hardcoded hash values in production

OPEN FILES
- Guardfile
- edit.mustache
- _form.html.erb

FOLDERS
- script_hash_test
  - app
    - assets
    - controllers
    - helpers
    - mailers
    - models
    - templates
      - articles
        - edit.mustache
      - layouts
    - views
  - bin
    - bundle
    - rails
    - rake
  - config
  - db
  - lib
  - log
  - public
  - test
  - tmp
  - vendor
  - .gitignore
  - .rvmrc
  - config.ru
  - Gemfile
  - Gemfile.lock
  - Guardfile
  - hash.tmp
  - in
  - inn
  - LICENSE
  - literal
  - Rakefile
  - README.md
  - README.rdoc

Guardfile | edit.mustache | _form.html.erb

```
1 guard 'rake', :task => 'secure_headers:generate_hashes', :task_args =>['true'] do
2   watch(%r{^app/.+\.(html\.erb|rhtml|mustache)$})
3 end
4
```

Line 4, Column 1 — Spaces: 2 — Plain Text