In the DOM, no one will hear you scream

A journey into the moldy layer
between HTML and JavaScript

A talk by Mario Heiderich
mario@cure53.de || @0x6D6172696F

Horst Görtz Institut
für IT-Sicherheit

Meta-Expert, Philanthropist, Visionary & Thought-Leader

- **Dr.-Ing. Mario Heiderich**
  - Researcher and Post-Doc, **R**uhr-**U**ni **B**ochum
    - PhD Thesis about Client Side Security and Defense
  - Founder of Cure53
    - Pentest- & Security-Firm located in Berlin
    - Consulting, Workshops, Trainings
    - „Simply the Best Company in the World“
  - Published Author and Speaker
    - Specialized on HTML5, DOM and SVG Security
    - JavaScript, XSS and Client Side Attacks
  - HTML5 Security Cheatsheet
  - **And DOMPurify!**
    - @0x6D6172696F
    - mario@cure53.de

Today's Menu

- The DOM (Document Object Model)
- Especially its weirder areas
  - Origin and Goals
  - History and first implementations
  - Traps and Pitfalls
  - Security Issues
  - Countermeasures against those
  - An Outlook
- No JavaScript-”Weirdness”
  - No `undefined==null` and so on
- We'll stick with the DOM itself - the “Layer Between™”
- Focus on security for modern web apps

Theodoros of Kyrene shows his mom a nasty Memory Leak

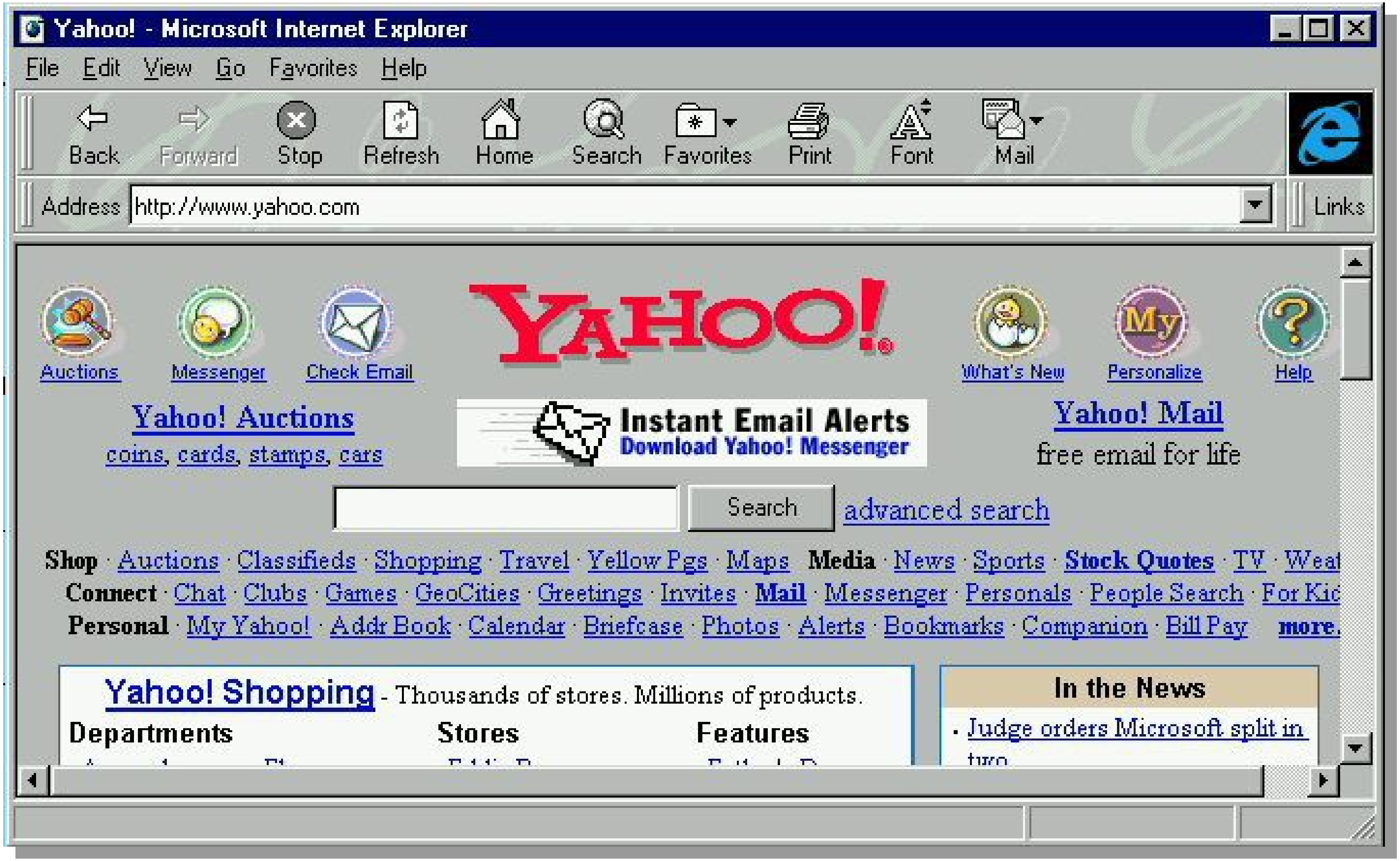
hgi
Horst Görtz Institut
für IT-Sicherheit

RUB

cure53

Ancient History

- The DOM as we know it today has made a very long way
- Baby steps were made as early as back in 1995
  - „Legacy-DOM“ or DOM Level 0
  - Implementations in Netscape 2.0 and MSIE 3.0
  - No actual standard. And why would there be any.
  - Partial documentation
  - No common denominator among browsers
  - JavaScript versus JScript
  - Poor on features, no actual feature-parity to HTML
- Goals of that early DOM?
  - Interactivity and easy element-access
- `document.forms[0].elements[0]`
- `document.bla.blubb`



Address: http://www.yahoo.com

Yahoo! Auctions
coins, cards, stamps, cars

Instant Email Alerts
Download Yahoo! Messenger

Yahoo! Mail
free email for life

advanced search

Shop · Auctions · Classifieds · Shopping · Travel · Yellow Pgs · Maps **Media** · News · Sports · **Stock Quotes** · TV · Weat
Connect · Chat · Clubs · Games · GeoCities · Greetings · Invites · **Mail** · Messenger · Personals · People Search · For Kic
Personal · My Yahoo! · Addr Book · Calendar · Briefcase · Photos · Alerts · Bookmarks · Companion · Bill Pay **more...**

Yahoo! Shopping - Thousands of stores. Millions of products.

Departments **Stores** **Features**

In the News

- Judge orders Microsoft split in two

The Intermediate-DOM

- After Legacy DOM there was a short intermediate phase
- The year we're in? 1997
- The browsers in control? MSIE and Netscape 4.0
- Implemented is the so called “Intermediate DOM”
- MSIE and Netscape place their bets on DHTML
  - „Dynamic HTML“
  - More APIs to influence HTML via JavaScript
  - But still no standard in sight
  - Any why would they, it's a browser war anyway
- So we're essentially talking about “DOM Level 0+”
- Still nothing spectacular, a niche in a niche

Now, DOM Level 1

- In the year 1998 DOM 1 reached recommendation status
- W3C DOM Level 1. slim but better than nothing
- After 4 years, finally something standard-like emerges
    - http://www.w3.org/TR/REC-DOM-Level-1/
    - Available components were „Core“ and „HTML“
    - “Naming Conventions”
    - “Document Structure”
    - “Case Sensitivity”
    - “Memory Management”
    - “Processing Instructions“
- Interfaces defined via IDL
    - Interface Description Language, Web IDL
- Still very XML-heavy, no trace of today's HTML
    - CDATA, Entities, Notations, etc. etc.

Conformity?

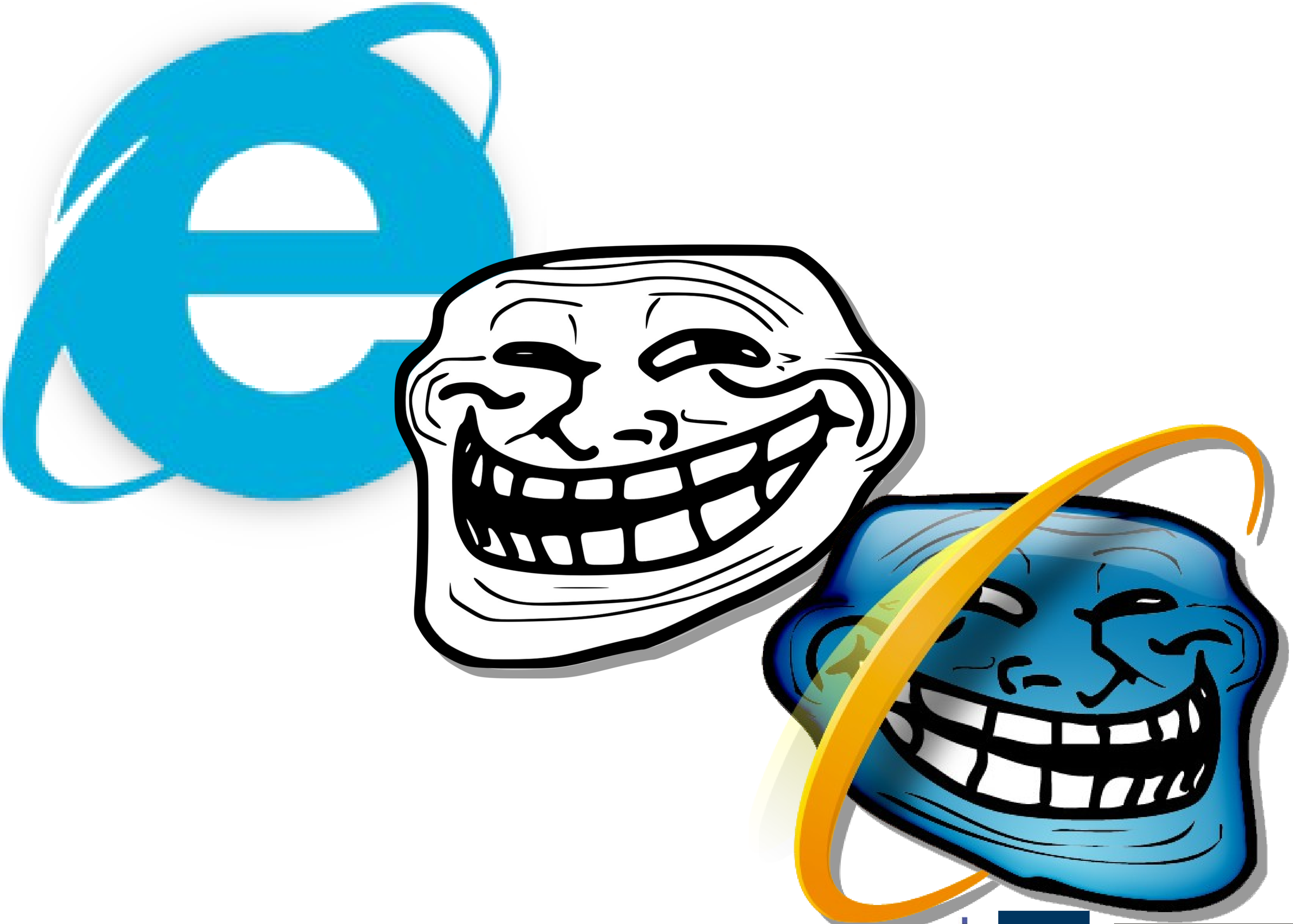
- What use is a standard if no one implements it?
- And did browser implement is?
  - Nooope. And, as said, why would they.
  - `document.all` in MSIE
  - `document.layers` in Netscape
  - `elm.innerHTML` – first in MSIE then copied all around
  - ActiveX and... `GeckoActiveXObject` *(okay, that got canceled)*
  - VBScript, the language from outer space
- MSIE5 shipped full DOM 1 Support. But tons of extras and deviations too
- Many of which are now also part of the standard
- JavaScript versus JScript again
  - Even today we witness relics of that time
  - `location('vbscript:msgbox(1)')`
  - `location.href = 'javascript:alert(1)'`

DOM Level 2

- Published by the W3C in late 2000
  - http://www.w3.org/TR/DOM-Level-2-Core/
- Enriched with the following modules
  - “Core”, „HTML“, „Events“, „Style“, „Views“ etc.
  - Better separation of the single satellite standards
  - For instance DOM Level 2 Events
  - http://www.w3.org/TR/DOM-Level-2-Events/
- Several small but important changes
  - `document.getElementById()` for all document types
  - Before that available HTML-only – alternative was „Traversal“ and ”Direct Access”
- Oh yes, and events of course
- „Something happens in case something occurs“
  - `document.createEvent()` etc.
- Otherwise stagnation, over at W3C the climate decreased
- Developers and Browser-Vendors wanted more. Much more.
- And so they just planned and built it in themselves.

Features in MSIE5

- A lot of things we consider hip these days
  - Favorites, MHTML, Data Islands, XHR, XDR
  - ActiveX, WD-XSL, Media Player, Toolbars
  - HTA, Conditional Compilation, Active Desktop
  - Cursor Capture, own Java VM, XMLDOM
  - Bidi-Text, Ruby Characters, Language Encoding
  - VML, SAMI, SMIL, CSS Filters, Page Transitions
  - DOM Behaviors, WebControls, HTML+TIME
  - Media Bar, Radio Bar, Persistence, HTC, TDC
  - Scriptable Editing, Viewlink Behaviors, DesignMode
- Many of those disappeared
- Some stayed though
- Others are hidden behind IE's “Docmodes”



hgi

Horst Görtz Institut
für IT-Sicherheit

RUB

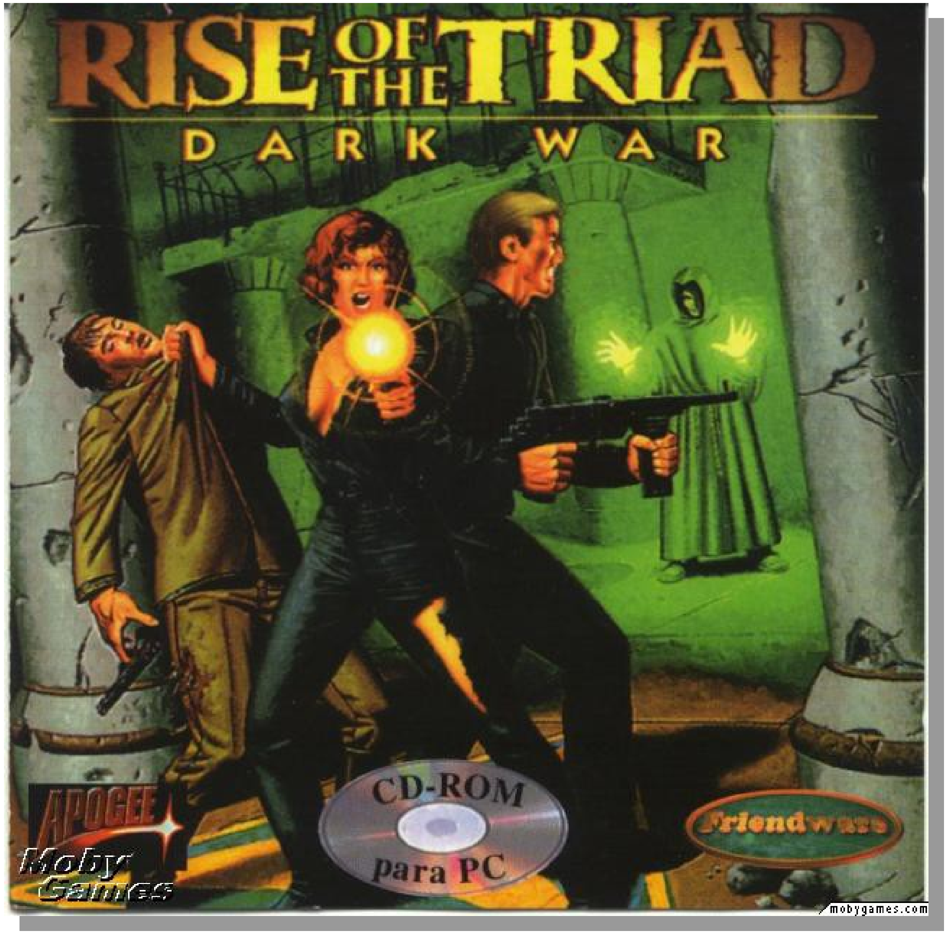
cure53

DOM Level 3

- The W3C continues moving slowly. Very slowly.
- DOM3 meanders into position to take off. Slowly.
- Specified in 2004, so now about ten years old
- Same year, the WHATWG was created and gained ground
  - Coincidence? Maybe. Maybe not.
  - No more slow-moving, XML-bound W3C?
  - Some great ideas by WHATWG, and some less ideal ones
  - Web Workers, Web Forms 2.0, “Living Standard”
- DOM3 is still XML-heavy
  - XML Serialization, XPath Support
- And finally Keyboard Events
  - „The DOM Level 2 Event specification **does not provide a key event module**. An event module designed for use with keyboard input devices will be included in a later version of the DOM specification.“

Rise of the Triad

- **Prototype**
  - First release in 2005
  - “Monkey Patching”, extending the DOM
  - Implements what's missed by developers
- **jQuery**
  - First release in August 2006
  - Fast and reliable access to DOM APIs
  - Avoiding browser-specific code
    - Conditional Comments, CSS Hacks, A Pis
    - Conditional Compilation
- **MooTools**
  - First release in September 2006
  - OOP in JavaScript
  - Extending the Element constructors
  - More control over HTML via JavaScript – yet another DOM so to say



The DOM Today

The DOM Today

- Specified by the W3C and others as DOM Level 4
- And also by WHATWG, and a bunch of other vendors
  - window.btoa() „DOM Level 0. Not part of any standard. Except of course http://www.whatwg.org/specs/...“
- „Many DOMs“, one goal: API between structure and logic
  - HTML DOM
    - http://www.w3.org/TR/dom/
    - http://dom.spec.whatwg.org/
  - SVG DOM
    - http://www.w3.org/TR/SVG/svgdom.html
    - http://www.w3.org/TR/SVG2/svgdom.html
  - PDF DOM
    - http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf
  - XML DOM
    - http://msdn.microsoft.com/en-us/library/hf9hbf87%28v=vs.110%29.aspx
  - MathML DOM
    - http://www.w3.org/TR/MathML2/chapter8.html
- And not to forget – many satellite-specs
  - http://www.w3.org/TR/#tr_DOM

And then JSMVCOMFG

- JavaScript Model-View-Controller Frameworks
- Many developers still yearn for more DOM features
- Web Components coming up slowly. Too slow?
- DOM itself to weak for large scale applications?
  - No programmatic templating yet
  - No clean separation of code and content
  - No good re-usability
  - Hard-to-use i18n
- So there's a trend towards JSMVC
  - Or jsMvvM or MVW or...
  - “Super-heroic Frameworks”
- Extend HTML's powers
- Lock people away the DOM
- Force-feed individual interfaces
  - JSMVC Security https://code.google.com/p/mustache-security/

But now let's get to it

- We have seen the following
  - The DOM developed over more than one decade
  - Meanwhile the API is huge
  - Sometimes simple and intuitive
  - Sometimes complex, counter intuitive and congested
  - Still, without the DOM, nothing moves in the modern web
- What we want to see now
  - Well, how about the parts where „no one can hear you scream“?
  - Where can we find behaviors that are risky
  - How can we spot those behaviors
  - And when does security come into play?
  - Maybe even a small „0ld-Day“ for illustration?
- **So, let's get started!**

String-to-Code

- The DOM is overflowing on ways to turns strings into code
  - Be it HTML or direct JavaScript
  - Some of them are classics
  - Other not too well known
  - Then others rather hidden
  - Result? Usually DOMXSS
- Let's have a look at a list of those
  - Just as a small warm-up
- And then have a look at more exotic cases

String-to-Code Table

- document.execCommand(x)
- elm.style.cssText
- Additional CSS Properties

  - location=x
  - location(x)
  - location.href=x
  - location.replace(x)
  - location.assign(x)
  - document.URL=x
  - location.protocol=x

- navigate(x)
- execScript(x)
- c.generateCRMFRequest(x)
- r.createContextualFragment(x)
- document.write(x)
- document.writeln(x)
- open(x)
- showModalDialog(x)
- showModelessDialog(x)

- elm.src=x
- elm.href=x
- elm.formAction=x
- elm.data=x
- elm.srcdoc=x
- elm.movie=x
- elm.value=x
- elm.values=x
- elm.to=x
- elm.on*=x
- elm.setAttribute(x)
- elm.setAttributeNS(x)
- elm.insertAdjacentHTML(x)
- elm.attributes.?.value=x

- eval(x)
- Function(x)()
- setTimeout(x)
- setInterval(x)
- setImmediate(x)
- msSetImmediate(x)

- elm.innerHTML=x
- elm.outerHTML=x
- elm.innerText=x
- elm.outerText=x
- elm.textContent=x
- elm.text=x

- $(x)
- $(elm).add(x)
- $(elm).append(x)
- $(elm).after(x)
- $(elm).before(x)
- $(elm).hhtml(x)
- $(elm).pprepend(x)
- $(elm).rreplaceWith(x)
- $(elm).wrap(x)
- $(elm).wwrapAll(x)

DOM Clobbering

- Not the most well-known attack technique
- Yet pretty effective if the stars are aligned well
  - Anyone knows the term already?
  - I think it was Gareth who coined it back then...
  - There's not too much documentation available
  - But the attacks can be fierce and hard to mitigate!
- So, who still remembers the site jibbering.com?
  - “Unsafe Names for HTML Form Controls”
  - http://jibbering.com/faq/names/
- And that is the very essence of DOM Clobbering

“Browsers also may add names and id's of other elements as properties to document, and sometimes to the global object (or an object above the global object in scope).

This non-standard behavior can result in replacement of properties on other objects. The problems it causes are discussed in detail.”

DOM Clobbering

```
<form id=foo>
        <input id=bar>
</form>

<script>
    alert(foo)
    alert(foo.bar)
</script>
```

DOM Clobbering

```
<form id=foo blafasel=xyz action=abc></form>

<script>
    alert(foo.blafasel)
    alert(foo.action)
</script>
```

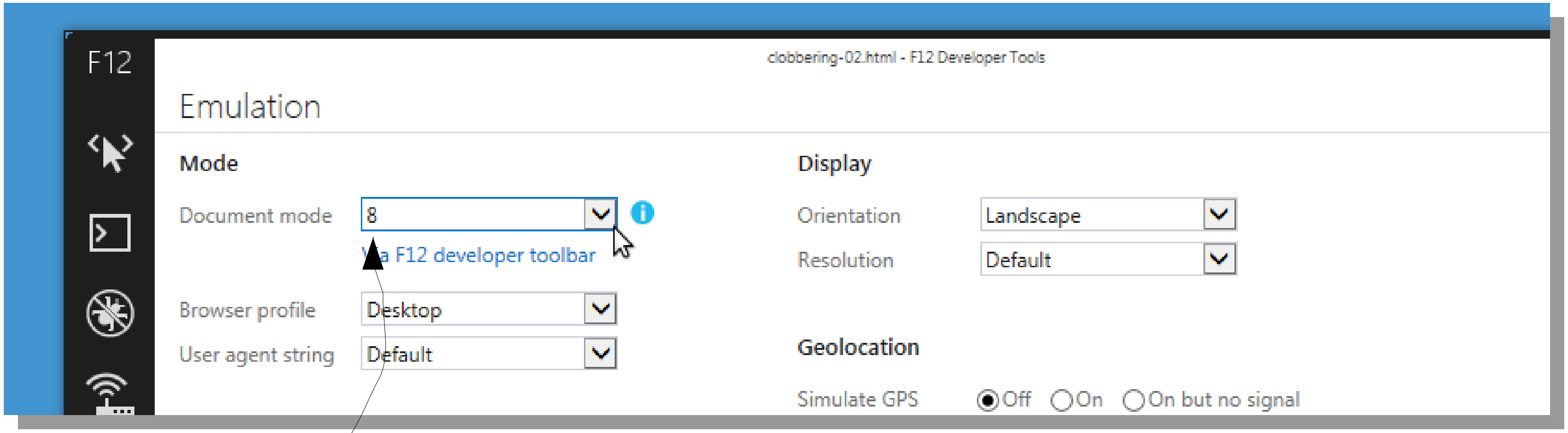
Soooo...

- Some attributes of FORM elements spawn global references
- And often, we can create child properties using certain attributes
  - And we can even assign strings to these child properties
- Jibbering.org calls these „Shortcut Accessors“
  - http://jibbering.com/faq/notes/form-access/#faShrt
- But it doesn't work in any case of course
- For most browsers, the attribute name must match an existing property in the element's constructor
  - Meaning – FORM knows `action` but not `blafasel`
  - So we can clobber `action` - but `blafasel` we cannot
  - Such a disappointment!

- And that's probably the case for each and every browser, right?
- **Riiight?**

No, not MSIE!

- On MSIE, we can also introduce children by using unknown attributes
- Meaning, properties that are unknown to the element's constructor
  - But only if the page is loaded in an older „docment mode“
  - Who still remembers document modes?
  - Exactly, the “solution” for compatibility problems and broken layouts
  - New MSIE, old engine, activate docmode via Header or META tag



clobbering-02.html - F12 Developer Tools

Emulation

Mode

Document mode: 8

Via F12 developer toolbar

Browser profile: Desktop

User agent string: Default

Display

Orientation: Landscape

Resolution: Default

Geolocation

Simulate GPS: ● Off ○ On ○ On but no signal

There,
IE8 Mode

Influencing Docmodes

- So, a page you want to clobber is in Edge-Mode?
- You can just load it in an Iframe in IE8-Mode
- It will adopt the docmode of the parent/top page
  - (X-Frame-Options protects: https://cure53.de/xfo-clickjacking.pdf)

```
<form id=abc def=123>
</form>
<script>
    alert(abd.def)
</script>
```

No dice

Yes dice :)

```
<meta
   http-equiv=x-ua-compatible
   content=IE=8
>
<iframe src=clobber.html>
```

And that means?

Yes, I am listening?

Yaaaay!

More Clobbering

```
<form id="blafasel"></form>
<script>
alert(blafasel)
</script>
```

```
<form id="foobar"></form>
<script>
foobar=1;alert(foobar)
</script>
```

```
<form id="blablubb"></form>
<script>
var blablubb=1; alert(blablubb)
</script>
```

```
<form id="honk"></form>
<script>
(function(){
    alert(honk)
})()
</script>
```

```
<form id="plonk"></form>
<script>
(function(plonk){
    alert(plonk)
})(1)
</script>
```

So, attackers can...

- Use harmless HTML to severely influence the DOM
  - For example to create new properties and child properties in the global scope
  - Overwrite existing variables
  - In case they have not been initialized
  - Or passed as an argument
- Well, that's fair enough
- But it's getting even better...

Again our friend MSIE

- Just for older versions
- But still...

```
<form id="document" cookie="123"></form>
<script>
alert(document.cookie)
</script>
```

```
<form id="location" href="javascript:alert(1)"></form>
<script>
alert(location.href)
</script>
```

DOM Clobbering Attack

- Now, let's have a look at an actual security bug
- It existed for years in the code of a popular RTE
- We're talking about the software called „CKEditor"

> *"The best web text editor for everyone"*
>
> *"World class quality"*
>
> *"High standard of quality"*

- Proper level of modesty, always good...
- **Let's watch a Demo (PoC below)**

```
<a href="plugins/preview/preview.html#<svg
onload=alert(1)>" id="_cke_htmlToLoad"
target="_blank">Click me for dolphins!</a>
```

The vulnerable Code

/plugins/preview/preview.html

```
<script>

var doc = document;
doc.open();
doc.write( window.opener._cke_htmlToLoad );
doc.close();

delete window.opener._cke_htmlToLoad;

</script>
```

To wrap it up...

- The attack works for the following reasons
  - We have a `document.write()`
  - We have implicit access to opener
  - We can influence a globally scoped „variable“
  - We actually have full (string) control via <a>+ `id`
  - <a> + `toString()` = Content of the `href` attribute
  - Encoding peculiarities for `window.location` help us
    - Some browsers encode special characters (Firefox)
    - Some do not (IE, Chrome, Safari, Opera, ...)
  - **Result: XSS via DOM Clobbering**

One Security Problem

- The whole things points at a general problem
- We do have great XSS filters on the server
  - HTMLPurifier, SafeHTML, AntiSamy etc.
- But we don't have much in the browser
  - Okay, MSIE has `toStaticHTML()`
  - Then we have XSS-Filters in the browser, IE, WebKitWebKit/Blink, NoScript
  - And there's a bunch of hacks and whacks
  - Sandboxed Iframes might be a way as well
  - Then jSanity.. but it never got released
- So we were like.. let's build something
- **CANNOT BE SO HARD RITE!!1**
- Just quickly write some client-side XSS filter

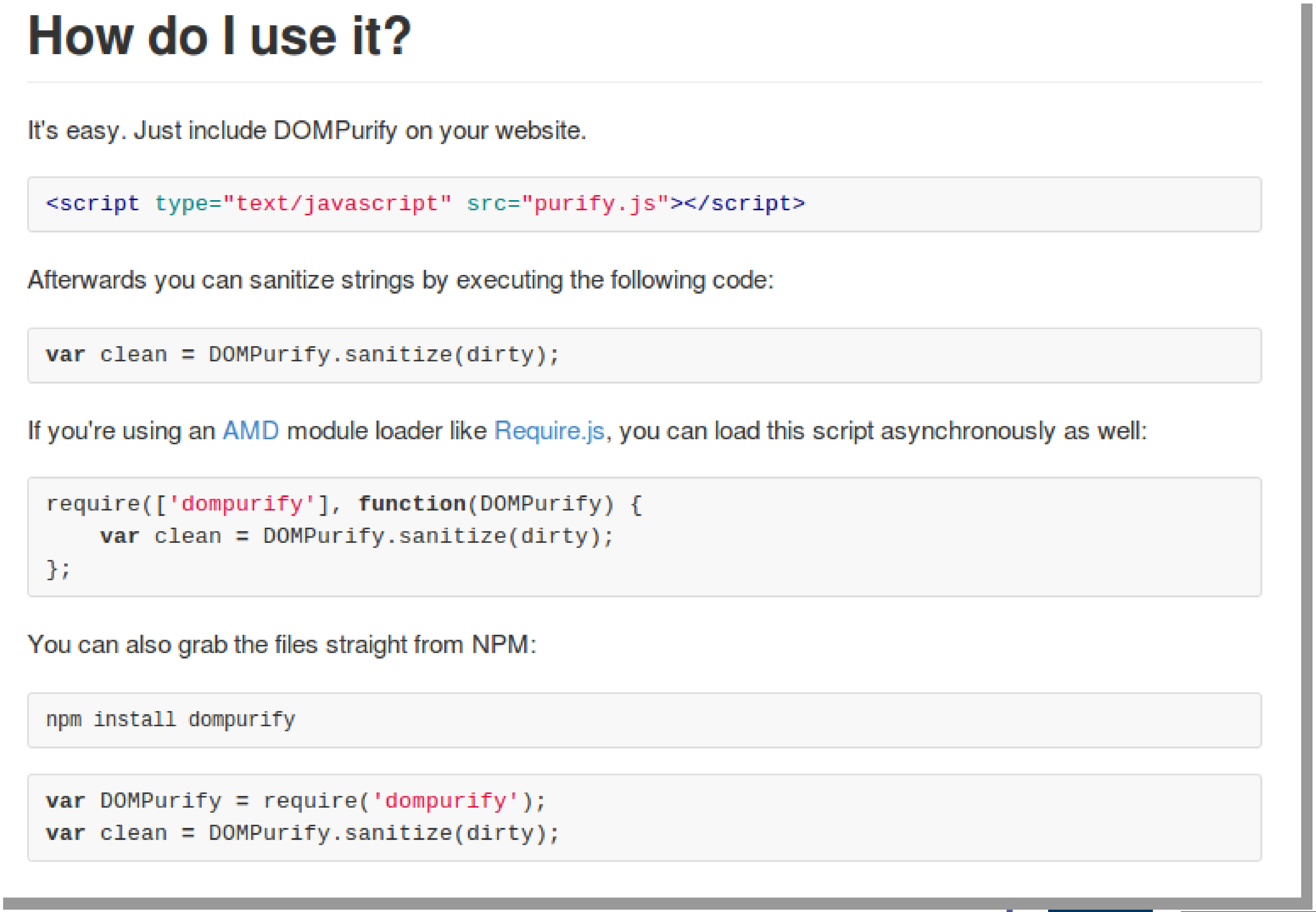
DOMPurify, a solution?

- So we need a new tool, let's write it
- And solve client-side issues where the happen
- In the client itself. Yeah!
  - XSS filter written in JavaScript, running on the DOM
  - Simple API. Dirty string in, clean string out
- Why in the client? Because of the „knowledge parity“!
  - Servers cannot solve XSS since they don't know the client
  - **This is fundamentally important! Always keep that in mind!**
  - The sever can only try to understand the client
  - And provide protection as good as possible. But **never** 100%
- And sometimes there is no server, then what?
  - Offline-Applications
  - Apps and Widgets
  - Web Crypto! Mailvelope for example, PGP in the browser

Again, because it's really so important.

Server-side XSS protection *cannot* guarantee 100% safety. It's *impossible* by design

DOMPurify API



How do I use it?

It's easy. Just include DOMPurify on your website.

```
<script type="text/javascript" src="purify.js"></script>
```

Afterwards you can sanitize strings by executing the following code:

```
var clean = DOMPurify.sanitize(dirty);
```

If you're using an AMD module loader like Require.js, you can load this script asynchronously as well:

```
require(['dompurify'], function(DOMPurify) {
    var clean = DOMPurify.sanitize(dirty);
};
```

You can also grab the files straight from NPM:

```
npm install dompurify
```

```
var DOMPurify = require('dompurify');
var clean = DOMPurify.sanitize(dirty);
```

Protect against XSS. Easy.

- DOMPurify tries to be as tolerant as possible
- Permit everything that doesn’t hurt. Literally everything.
- Very generous white-list
  - Known as secure? Is allowed!
  - Not sure or unknown? Blocked!
- **Available for HTML, SVG and MathML!**
  - And whatever ?ML people might come up with
- Even works with Shadow DOM, we'll see that later
- Secure default, Config-API for customizations
- Technological base for the tool is as follows:
  - `document.implementation.createHTMLDocument()`
  - `document.createNodeIterator()`
  - `document.removeChild()`
  - `document.removeAttributeNode()`
  - Final serialization and return of the sanitized string. Or DOM.

The DOM, an old Buddy.

- That all sounds quite easy, right?
- XSS solved in the client. Shwoops, done.
- But the DOM decided to take revenge on us. Back-stabbed us.

- So, a security library must be able to withstand attacks
  - And the attacker can use whatever she finds in the DOM
  - Peculiarities turn weaknesses, weaknesses turn vulnerabilities
  - And vulnerabilities turn into exploits
  - And that happened.
- The work on DOMPurify showed us, what incredible mess the DOM really is.
- Let's now have a close look at that...

1. DOM Clobbering

- The DOMPurify Pre-Alpha was tested thoroughly before release
- And broken several times. Painfully broken too.
- But the first bypasses had nothing to do with XSS
- But with the DOM, its behavior and the weirdness to it
  - Which eventually leads to XSS as we already saw
- So, ladies and gentlemen, what would this snippet of markup do?

```
<div onclick=alert(0)>
 <form onsubmit=alert(1)>
   <input name=parentNode>123
 </form>
</div>
```

1. The Effect

- Our code used the property `parentNode`, see below
- This property however does not exist anymore in its original form
- It got overwritten by its own child element!
  - `child.parentNode === child // wtf, DOM!`
- Unfortunately we need the `parentNode` property
- So we need to... authenticate and verify `parentNode`
- Is it that `child.parentNode === child`? Yes? Potential attack!

```
/* Remove element if anything prohibits its presence */
currentNode.parentNode.removeChild(currentNode);
```

2. “Clobbering” Attributes

- That was already pretty nasty
- But it gets a lot worse
- As a security-library we of course have to cover HTML attributes too
- And, if necessary, safely remove them to prevent XSS
- Now let's have a look at the following bypass

```
<form onmouseover='alert(1)'>
   <input name="attributes">
   <input name="attributes">
</form>
```

2. The Effect

```
for (var attr = elm.attributes.length-1; attr >= 0; attr--) {

         tmp = elm.attributes[attr];
         clobbering = false;
         elm.removeAttribute(elm.attributes[attr].name);

...
```

- Our code iterated over `attributes` to find out which ones exist
- And then to check their values
- But what if `attributes` is suddenly an HTML element?
- Then the code breaks, XSS is nigh
- So we have to go and check again
  - `if(typeof elem.attributes.item === 'function')` …
  - **Looks okay, right?**

2. Yeah, well...

- Our checks looked nice at first, but they were rubbish!
- Because there was another bypass!

```
<form onmouseover='alert(1)'>
   <input name="attributes">
   <input name="attributes">
</form>
```

- Now, the property `attributes` consists of ***two*** HTML elements
- And therefore it's a `NodeCollection`
- Which then again has the method `items()` exposed
- XSS! Dammit! So we need an even better check!

3. And it goes on like that...

- We learned that iterating is not as easy as it seems
- In the early phases on DOMPurify, we saw weird artifacts
  - Element has three attributes, two were removed
  - Went great for one. Then others turned invisible. And were not caught by our loop
  - All fine we thought, wrote the element back to the DOM
  - And „flooop“, the invisible attribute was back!

```
<div wow=removeme onmouseover=alert(1)>text
```

3. Gotta go backwards

- We have to remove attributes „backwards“. So starting with last and iterating on to the first
- Otherwise the browser has to re-sort! And thereby the index breaks and we have invisible attributes
- Invisible, but still there.

```
// wrong
for (var i = 0; i <= elm.attributes.length; i++) {
         elm.removeAttribute(elm.attributes[i].name);

// right
for (var attr = elm.attributes.length-1; attr >= 0; attr--) {
         elm.removeAttribute(elm.attributes[attr].name);
```

4. Document Clobbering

- Another trick that was used against us was evil images
- DOM Clobbering at its best, look at this!

```
<img src=bla name=getElementByID>

<image name=activeElement><svg onload=alert(1)>

<image name=body>
<img src=x><svg onload=alert(1); autofocus>,
<keygen onfocus=alert(1); autofocus>
```

5. Mutations or mXSS

- Again, **mXSS** is a huge issue, also in modern browsers
- We know that some properties get mutated and trigger XSS, invisible to the server
  - http://cure53.de/fp170.pdf
- Among those properties are `innerHTML` or `textContent`, `cssText`
- and many others
- And again, DOMPurify could be bypassed using those tricks

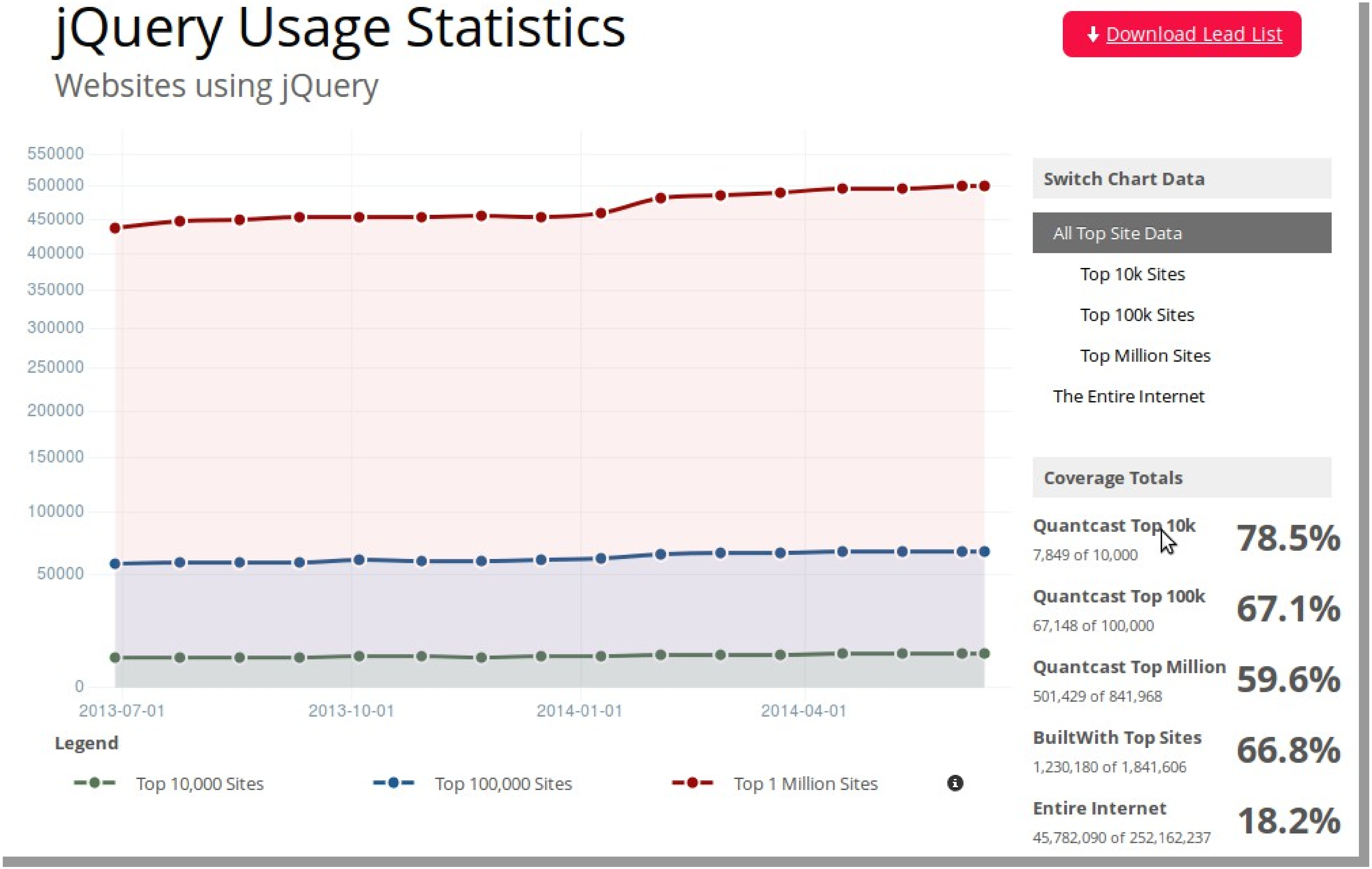
```
<listing>
&lt;img onerror=\"alert(1);//\" src=1&gt;<t t></listing>

<img src=x id/=' onerror=alert(1)//'>

123<a href='\u2028javascript:alert(1)'>I am a dolphin too!</a>
```

Security in the DOM?

- Doesn't really exist. Yet. We're getting there though!
- The following need to be kept in mind
  - DOM Clobbering, verification of properties
  - Overwritten and deactivated methods
  - Mutating values, mXSS
  - Protocol-Handlers using Unicode (weird Chrome bug)
  - Iteration in the right order
  - Verification of changes. Verification all the time
  - Proper reaction to anomalies
- **With DOMPurify we came quite far**
- But there's no 100% security yet
- And then there's still jQuery and friend, oh noez!
- And that's we we start at zero again. F****g jQuery!!1



jQuery Usage Statistics

Websites using jQuery

Coverage Totals

Quantcast Top 10k 7,849 of 10,000	**78.5%**
Quantcast Top 100k 67,148 of 100,000	**67.1%**
Quantcast Top Million 501,429 of 841,968	**59.6%**
BuiltWith Top Sites 1,230,180 of 1,841,606	**66.8%**
Entire Internet 45,782,090 of 252,162,237	**18.2%**

Facts

- jQuery is obviously used... quite a lot
  - About a fifth of all websites worldwide. A fifth!
- jQuery haunted by „Ghosts of XSS-mas Past“
  - Remember the debacle around `$(location.hash)`
  - Or `$('<svg onload=alert(1)>')`
  - The $-Factory, that not only selects and wraps but builds a DOM
  - And of all properties uses `innerHTML` and a DIV to map
- But it gets worse
- Let's have a look at the following attack vector

```
<option><style></option></select><b><img src=xx:
onerror=alert(1)></style></option>
```

And now what?

- Technically the vector is harmless. Cannot execute JavaScript
- And doesn't. And shouldn't.
- But once jQuery is present, things change because jQuery is „smart“ and wraps for conformity

```
// We have to close these tags to support XHTML (#13200)
wrapMap = {

    // Support: IE 9
    option: [ 1, "<select multiple='multiple'>", "</select>" ],

    thead: [ 1, "<table>", "</table>" ],
    col: [ 2, "<table><colgroup>", "</colgroup></table>" ],
    tr: [ 2, "<table><tbody>", "</tbody></table>" ],
    td: [ 3, "<table><tbody><tr>", "</tr></tbody></table>" ],

    _default: [ 0, "", "" ]
};
```

So?

- Now, our harmless HTML string turns into something very much different
- Look at this!

```
// Original
<option><style></option></select><b><img src=xx:
onerror=alert(1)></style></option>

// Result
<select multiple="multiple">
    <option><style></style></option>
</select>
<b>
    <img src="xx:" onerror="alert(1)" />
</b>
```

And there's even more...

- Thanks, jQuery, for the night shifts.
- DOMPurify now has a „Safe for jQuery“ mode
- But similar craziness can be done using the Shadow DOM
- With the new `<template>` element for instance
- Although this element technically has child element, we cannot just iterate over them. Because they are stored on `elm.content`.

```
<template id="tpl">
   <b>Heya!</b>
</template>

<script>
tpl.childNodes // Is empty, no child nodes
tpl.content.childNodes // Ah! There's our element!
</script>
```

Protect thy selves

- So, what can we do to protect ourselves?
- At the server-side level
  - Classic XSS „protection“ is not enough
  - ID and NAME have to be removed from user-generated markup
  - CLASS can get dangerous, when MVC are mixed in
  - Don't even build black-lists, White-lists are the only working approach
- At the client-side level
  - Clobbering is the biggest risk so far
  - It's easy to get a fresh DOM but hard to keep it reliable
  - Clobbering even happens in `document.implementation`
- Classic XSS Bugs will disappear in the next years
- Direct and indirect attacks against the DOM will become more prevalent
- So better get on track right now!
- The „XSS N1nja L33t Haxor bounty“ party is gonna be over soon

Conclusion

- Proper DOM security is hard
- Understanding the DOM is often hard as well
  - Traversal fails, transactions fail
  - Elements disappear, new elements pop up
- Without a string JavaScript/DOM Debugger you won't get far
- Browsers still do their own thing here and there
- However, first baby-steps were made
  - Documentation, Libraries, Browsers actually fix standard deviations
  - https://github.com/cure53/DOMPurify
  - https://github.com/cure53/jPurify
- Still, we kind of need a community wiki
  - And collect all those crazy artifacts in one place
  - And discuss the security implications
  - Maybe this? https://github.com/cure53/xss-challenge-wiki
- There's new features coming every day
- And the DOM develops fast(er than anything else in the WWW)

The End

- Question?
- Comments?
- Thanks a lot!
- And special thanks to all contributors and breakers of DOMPurify!