Im DOM hört Dich keiner schreien

Eine Reise in die gruselige Schicht
zwischen HTML und JavaScript

Eine Präsentation von Mario Heiderich
mario@cure53.de || @0x6D6172696F

hgi
Horst Görtz Institut
für IT-Sicherheit
RUB
cure53

Meta-Experte, Visionary & Thought-Leader 3.0

- **Dr.-Ing. Mario Heiderich**
  - Forscher und Post-Doc, **R**uhr-**U**ni **B**ochum
    - PhD Thesis über Client Side Security und Defense
  - Gründer von Cure53
    - Pentest- & Security-Firma in Berlin
    - Consulting, Workshops, Trainings
    - „Simply the Best Company of the World“
  - Publizierter Autor und Speaker
    - Spezialisiert auf HTML5, DOM und SVG Security
    - JavaScript, XSS und Client Side Attacks
  - HTML5 Security Cheatsheet
  - **Und etwas neues!**
    - @0x6D6172696F
    - mario@cure53.de

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Unser Heutiges Programm

- Das DOM (Document Object Model)
  - *%insert obvious cologne joke here*
- DOM und seine Tücken & Schrecken
  - Ursprung und Zielsetzung
  - Geschichte und Entstehung
  - Tücken und Gräuel
  - Sicherheitsprobleme
  - Gegenmaßnahmen
  - Weitere Tücken und Wunderlichkeiten
  - Ausblick
- Keine JavaScript-”Weirdness”
  - Kein `undefined==null` und so weiter
- Wir bleiben im DOM, der “Schicht dazwischen™”
- Fokus liegt auf Security für moderne Webapps

Düsseldorf,
Perle des Ruhrgebiets

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Theodoros von Kyrene zeigt seiner Mutter ein Memory Leak

Graue Vorgeschichte

- Das DOM wie wir es heute kennen hat eine lange Reise hinter sich
- Erste Schritte wurden im Jahre 1995 gemacht
  - „Legacy-DOM“ oder DOM Level 0
  - Implementationen in Netscape 2.0 und MSIE 3.0
  - Kein Standard. Wieso auch.
  - Partielle Dokumentation
  - Kein gemeinsamer Nenner zwischen Browsern
  - JavaScript versus JScript
  - Kaum Features, keine Feature-Parität zu HTML
- Zielsetzung des DOM?
  - Interaktivität und einfacher Element-Zugriff
- `document.forms[0].elements[0]`
- `document.bla.blubb`

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Das Übergangs-DOM

- Nach dem Legacy DOM gab es eine kurze Übergangsphase
- Das Jahr? 1997
- Die Browser? MSIE und Netscape 4.0
- Implementiert ist das “Intermediate DOM”
- MSIE und Netscape setzen DHTML
  - „Dynamic HTML“
  - Mehr APIs um HTML in JavaScript zu beeinflussen
  - Immer noch kein Standard
  - Wieso auch, ist ja schließlich Browserkrieg
- Also im wesentlichen “DOM Level 0+”

DOM Level 1

- Im Jahre 1998 wurde vom W3C DOM Level 1 empfohlen. Sehr schlank aber immerhin etwas
  - Erstmals, seit circa 4 Jahren der Ansatz eines Standards
    - http://www.w3.org/TR/REC-DOM-Level-1/
  - Komponenten „Core“ und „HTML“
  - “Naming Conventions”
  - “Document Structure”
  - “Case Sensitivity”
  - “Memory Management”
  - “Processing Instructions“
- Interfaces definiert via IDL
  - Interface Description Langauge, Web IDL
- Noch sehr XML-lastig, keine Spur von heutigem HTML
  - CDATA, Entities, Notations, etc. etc.

Konformität?

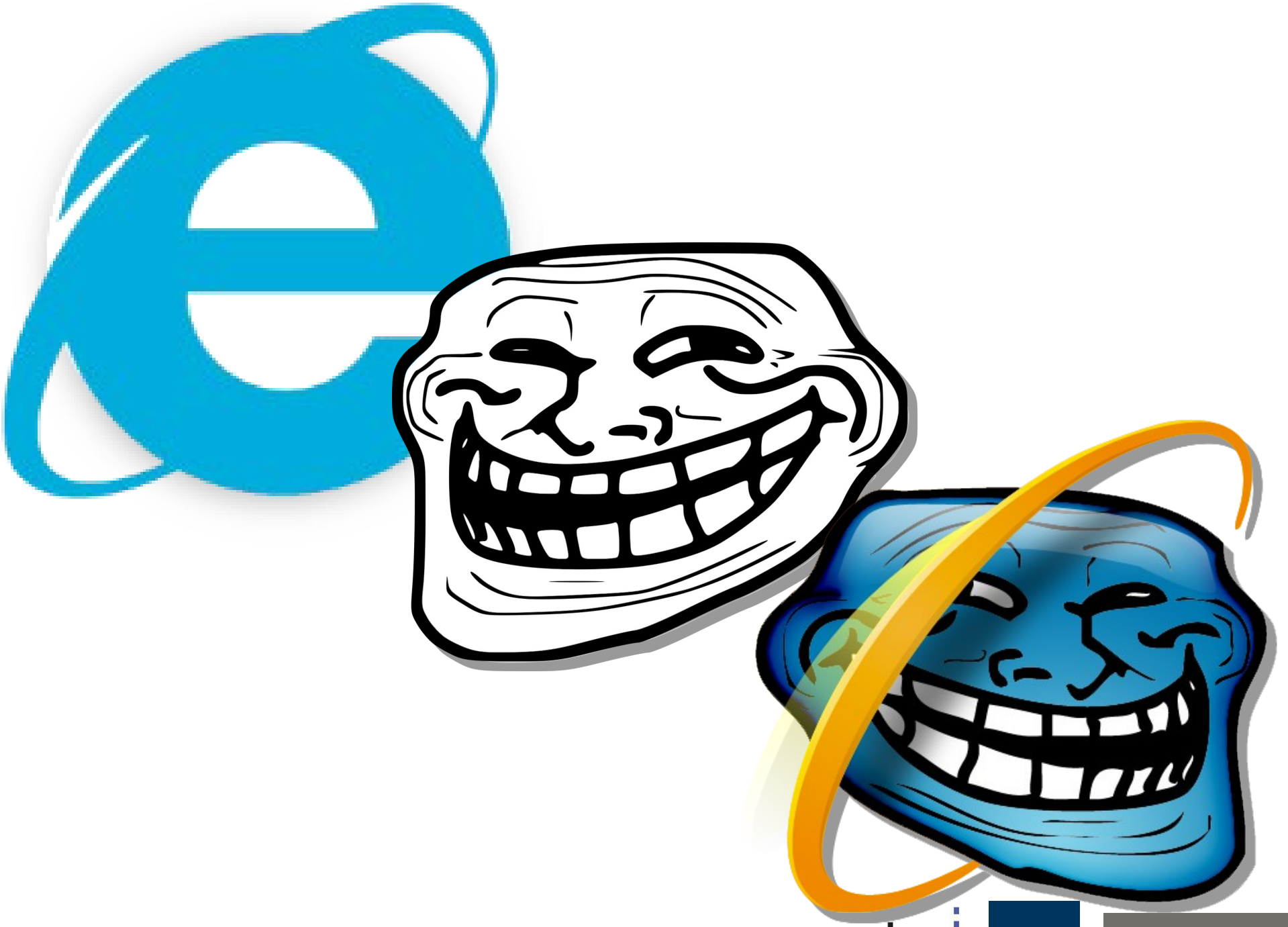
- Was nutzt der Standard, wenn sich keiner dran hält...
- Haben sich die Browser dran gehalten?
  - Nö. Warum auch.
  - `document.all` im MSIE
  - `document.layers` in Netscape
  - `elm.innerHTML` – später von allen kopiert
  - ActiveX und... `GeckoActiveXObject` *(okay, daraus wurde nichts)*
  - VBScript, die Sprache von einem anderen Stern
- MSIE5 hatte vollen DOM 1 Support. Aber auch tonnenweise Extras und Abweichungen. Von denen viele nun Teil des Standards sind
- JavaScript versus JScript
  - Selbst heute sehen wir noch Relikte aus dieser Zeit
  - `location('vbscript:msgbox(1)')`
  - `location.href = 'javascript:alert(1)'`

DOM Level 2

- Verabschiedet vom W3C im Jahre 2000
  - http://www.w3.org/TR/DOM-Level-2-Core/
- Erstmals mit den Modulen “Core”, „HTML“, „Events“, „Style“, „Views“ etc. angereichert
  - Bessere Trennung von Satellitenstandards
  - Zum Beispiel DOM2 Events
  - http://www.w3.org/TR/DOM-Level-2-Events/
- Eine fundamentale Neuerung
  - `document.getElementById()` für alle Dokumenttypen
  - Zuvor nur in HTML - sonst „Traversal“ und ”Direktzugriff”
- Ach ja, und Events natürlich
  - `document.createEvent()` etc.
- Ansonsten eher Stagnation, im W3C regte sich erster Unmut
- Entwickler und Browserhersteller wollten aber mehr
- Und haben's also selber gebaut, am Standard vorbei. Super.

Features im MSIE5

- Überraschend viel, was heute als neu gilt
  - Favorites, MHTML, Data Islands, XHR, XDR
  - ActiveX, WD-XSL, Media Player, Toolbars
  - HTA, Conditional Compilation, Active Desktop
  - Cursor Capture, eigene 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
- Einiges davon ist heute wieder verschwunden
- Anderes hinter alten “Docmodes” versteckt



hgi
Horst Görtz Institut
für IT-Sicherheit

RUB

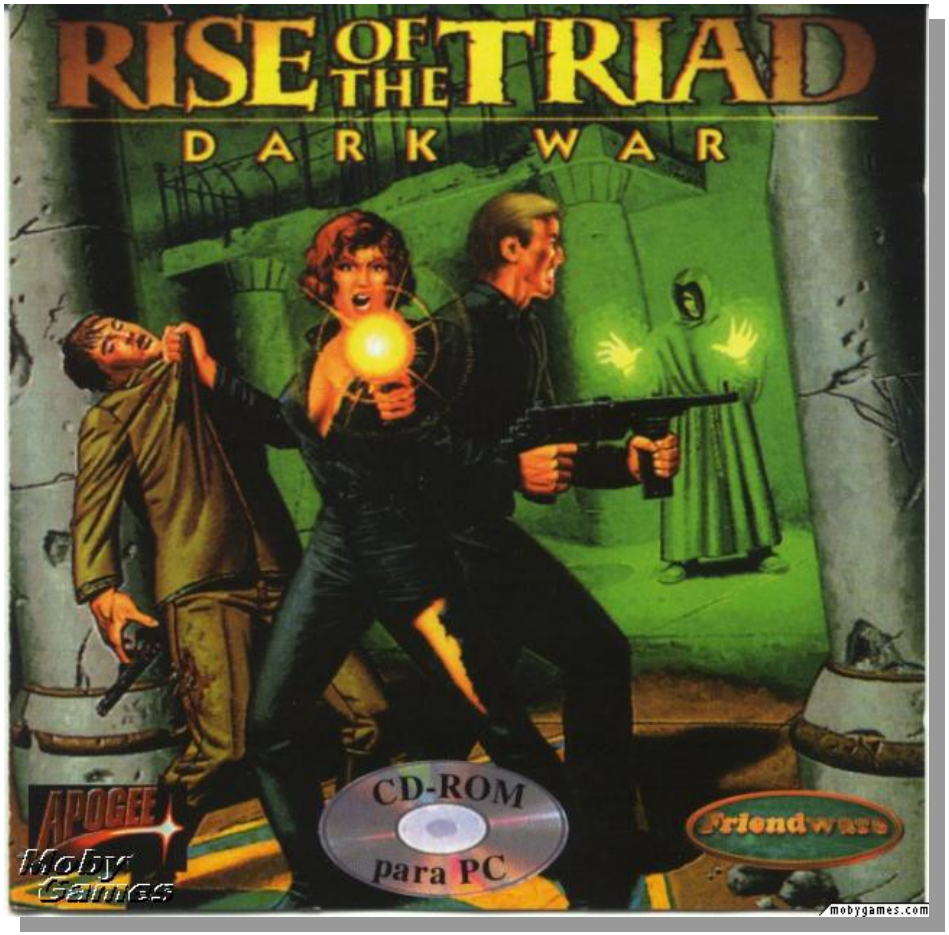
cure53

DOM Level 3

- Das W3C bewegte sich unbeirrt im Schneckentempo weiter
- DOM3 mäandert sich in die Startlöcher
- Spezifiziert im Jahre 2004, also gut zehn Jahre alt
- Das gleiche Jahr, in dem die WHATWG aktiv wurde
  - Zufall?
  - Keine Lust mehr auf langsames W3C
  - Mit guten Ideen und weniger guten Ideen
  - Web Workers, Web Forms 2.0, “Living Standard”
- DOM3 nach wie vor mit großem XML Anteil
  - XML Serialization, XPath Unterstützung
  - Keyboard Events

Rise of the Triad

- **Prototype**
  - Erster Release im Jahre 2005
  - “Monkey Patching”, Erweiterung des DOMs
  - Was nicht da ist bauen wir selber
- **jQuery**
  - Erster Release im August 2006
  - Einheitlicher Zugriff auf DOM APIs
  - Weg von Browser-spezifischem Code
    - Conditional Comments, CSS Hacks, Conditional Compliation
- **MooTools**
  - Erster Release im September 2006
  - Objektorientierung in JavaScript
  - Erweiterung des Element Konstruktors
  - Mehr Kontrolle über HTML via JavaScript – ein eigenes DOM sozusagen



DOM Heute

DOM Heute

- Spezifiziert unter anderem vom W3C, DOM Level 4
- Und von der WHATWG, oder bestimmten Vendors
  - „DOM Level 0. Not part of any standard. Except of course http://www.whatwg.org/specs/...“
- „Viele DOMs“, ein Ziel: API zwischen Code und Content
  - 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
- Nicht zu vergessen – viele Satelliten-Spezifkationen
  - http://www.w3.org/TR/#tr_DOM

Und dann noch JSMVCOMFG

- JavaScript Model-View-Controller Frameworks
- Vielen reicht die Funktionalität noch immer nicht aus
- Web Components langsam im Kommen. Zu langsam?
- DOM zu schwach für große Applikationen?
  - Kein programmatisches Templating
  - Keine saubere Trennung von Code und Inhalt
  - Hakelige Internationalisierung
- Daher ein Trend in Richtung JSMVC
  - Oder jsMvvM oder MVW
  - “Superheroic Frameworks”
- HTML Erweitern
- Das DOM unzugänglich machen
- Eigene Interfaces erzwingen
  - JSMVC Security https://code.google.com/p/mustache-security/

Willst Du eine Website bau'n,
die recht schlank und schick ist?
❤❤❤
Schau dir erst das Framework an,
ob es nicht zu dick ist!

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Aber nun zum Thema!

- Wir haben gesehen
  - Das DOM hat sich über Jahrzehnte entwickelt
  - Mittlerweile ist die API riesig
  - Manchmal einfach, manchmal komplex
  - Ohne DOM geht nix
- Wir wollen gerne sehen
  - Wo sind Bereiche, in denen „dich niemand schreien hört“
  - Wo treten Verhaltensweisen auf, die deduktiv sind
  - Wo und wie findet man solche Bereiche
  - Und was hat das ganze mit Security zu tun
  - Vielleicht noch 'nen kleinen „0-Day“?
- **Fangen wir also an!**

String-to-Code

- Das DOM ist voll von Möglichkeiten, Strings in Code umzuwandeln
  - Einige davon sind Klassiker,
  - Einige davon einigermaßen bekannt
  - Andere eher unbekannt
  - Häufiges Resultat? DOMXSS
- Schauen wir uns eine Liste selbiger an
  - Zur Einstimmung auf das Thema
- Und kommen dann zu den exotischeren Fällen

String-to-Code Tabelle

- document.execCommand(x)
- elm.style.cssText
- Weitere 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).html(x)
- $(elm).prepend(x)
- $(elm).replaceWith(x)
- $(elm).wrap(x)
- $(elm).wrapAll(x)

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

DOM Clobbering

- Weitgehend unbekannte Angriffstechnik
  - Kennt jemand den Begriff?
  - Ein wenig Doku dazu gibt es, aber nicht viel
- Wer kann sich an jibbering.com erinnern?
  - “Unsafe Names for HTML Form Controls”
  - http://jibbering.com/faq/names/
- Genau darum geht’s bei 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>
```

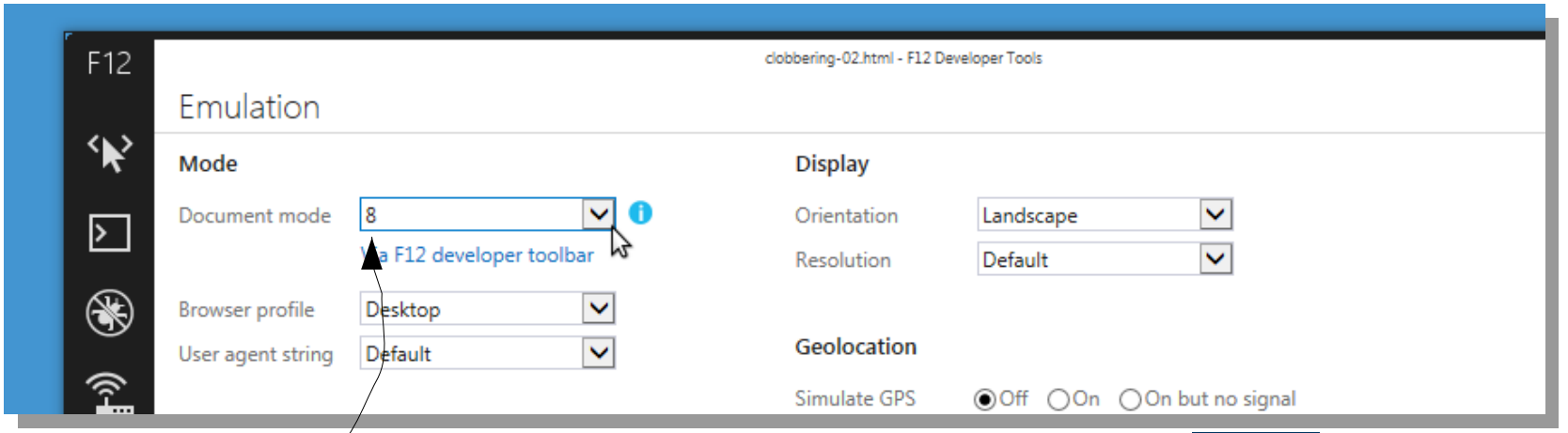
Also...

- Bestimmte Attribute in FORM legen globale Referenzen an
- Oft können mit Kind-Elementen Eigenschaften angelegt werden
- Mit Attributen können ebenfalls Eigenschaften angelegt werden
- Jibbering.org nennt diese „Shortcut Accessors“
  - http://jibbering.com/faq/notes/form-access/#faShrt
- Aber nicht immer!
- Keine dem jeweiligen Konstruktor unbekannten Attribute
  - Sprich – FORM kennt `action` aber nicht `blafasel`
  - Daher ist `action` belegt – aber `blafasel` nicht

- Das sollte bei allen Browsern so sein
- **Oder?**

Nicht so im MSIE!

- Da klappen auch unbekannte Attribute
- Beziehungsweise Eigenschaften, die der Konstruktor nicht kennt
  - Aber nur, wenn die Seite in älteren “Docmodes” geladen wird.
  - Wer weiß noch, was Docmodes sind?
  - Genau, die “Lösung” für Kompatibilitätsprobleme
  - Neuer IE, alte Engine, Aktivierung via Header oder META Tag



Da, IE8 Modus

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Docmodes beeinflussen

- Ziel-Seite läuft im Edge-Mode?
- Einfach mit einer anderen Seite im IE8-Mode “Iframen”
  - (X-Frame-Options als Schutz, https://cure53.de/xfo-clickjacking.pdf)

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

Geht nicht

Klappt prima

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

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Und das bedeutet?

Yaaaay!

Mehr 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>
```

Angreifer können...

- Mit harmlosem HTML das DOM beeinflussen
  - Neue Objekte und Eigenschaften im globalen Scope anlegen
  - Variablen überschreiben
  - Solange diese nicht initialisiert wurden
  - Oder als Argument übergeben wurden
- Und es kommt noch besser...

Mal wieder der IE

- Immerhin nur ältere Versionen
- Aber dennoch...

```
<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 “0-Day”

- Existierende Sicherheitslücke
  - “0-Day” im CKEditor

    “The best web text editor for everyone”

    “World class quality”

    “High standard of quality”

- Ein gutes Maß an Bescheidenheit
- **Demo…**

Verwundbarer 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>
```

Zusammengefasst...

- Der Angriff funktioniert aus folgenden Gründen
  - Wir haben ein `document.write()`
  - Wir haben Zugriff auf `opener`
  - Wir haben zugriff auf eine globale Variable
  - Wir haben Kontrolle über diese via `<a>+ id`
  - `<a>` + `toString()` = Inhalt des `href` Attributs
  - Kodierung in `window.location` Eigenschaften?
    - Manche Browser kodieren Sonderzeichen (Firefox)
    - Manche nicht (IE, Chrome, Safari, Opera, ...)
  - **Resultat: XSS via DOM Clobbering**

Ein Sicherheitsproblem

- Das Ganze zeigt ein wichtiges Problem auf
- Wir haben gute XSS Filter für den Server
  - HTMLPurifier, SafeHTML, AntiSamy etc.
- Aber wir haben nix im Browser
  - Okay, MSIE hat `toStaticHTML()`
  - Dann noch die XSS-Filter *im* Browser, IE, WebKit/Blink, NoScript
  - Und es gibt andere Hacks
  - Sandboxed Iframes sind manchmal ein Weg
- Wir dachten uns, wir bau'n mal was
- Kann ja nicht so schwer sein
- Mal eben schnell so mit Links

DOMPurify als Lösung?

- Da muss ein Tool her - dachten wir uns so...
- Client-Probleme im Client lösen
  - XSS Filter komplett in JavaScript
  - Unsicherer String rein, sicherer String raus. Fertig.
- Warum? Weil wir sonst ein Wissensproblem bekommen!
  - “Was der Bauer nicht weiss...”
  - Server können XSS nicht lösen da sie den Client nicht kennen
  - Der Server kann nur versuchen, den Browser/Client zu verstehen
  - Und darauf basierend Schutz zu bieten. So gut wie möglich
- Oft haben wir auch gar keinen Server
  - Offline-Applikationen
  - Apps und Widgets
  - Crypto! Mailvelope zum Beispiel, PGP im Browser

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);
```

Schutz vor XSS. Easy.

- DOMPurify versucht, so tolerant wie möglich zu sein
- Alles erlauben was nicht weh tut. Alles.
- Sehr großzügige White-List
  - Alles was als sicher bekannt ist darf durch
  - Alles andere wird entfernt
- **Und das für HTML, SVG und MathML!**
- Und sogar Shadow DOM, sehen wir später noch
- Sicherer Default, Config-API für Anpassungen
- Technologische Basis des Filters ist wie folgt:
  - `document.implementation.createHTMLDocument()`
  - `document.createNodeIterator()`
  - `document.removeChild()`
  - `document.removeAttributeNode()`
  - Finale Serialisierung und Ausgabe

Das DOM, ein alter Freund?

- Klingt eigentlich alles ganz einfach
- Aber wir haben die Rechnung ohne das DOM gemacht

- Eine Security Library muss Angriffen standhalten können
  - Angriffe können auf die Tücken des gesamten DOM zurückgreifen
  - Wunderlichkeiten können zu Schwächen, Schwächen zu Sicherheitslücken werden
  - Harmlose Features können in Kombination mit anderen Features zum Exploit werden
- Die Arbeit an DOMPurify hat klargemacht, wie gruselig das DOM wirklich ist!
- Schauen wir uns das einmal an...

1. DOM Clobbering

- Die DOMPurify Pre-Alpha wurde gründlich getestet
  - Und mehrfach gebrochen. Böse gebrochen.
- Die ersten Bypasses hatten nichts mit XSS zu tun
- Sondern mit dem DOM und seinem Verhalten
  - Was dann zu XSS führte
- Meine Damen und Herren,
  was könnte dieser Beispielvektor anrichten?

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

1. Die Auswirkung

- Unser Code nutzt `parentNode`, siehe unten
- Diese Eigenschaft existiert aber nicht mehr wie gewünscht
- Sondern ist vom eigenen Kindelement überschrieben!
  - `child.parentNode === child // wtf, DOM!`
- Leider wird `parentNode` aber benötigt
- Wir müssen also `parentNode` authentifizieren
- Ist `child.parentNode === child`? Ja? Potenzieller Angriff!

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

2. Attribute “clobbern”

- Es kommt aber noch viel schlimmer
- Als Security-Library müssen wir natürlich auch Attribute absichern
- Und notfalls entfernen
- Schauen wir uns nun folgenden Bypass an

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

2. Die Auswirkung

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

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

...
```

- Unser Code iteriert über `attributes` um herauszufinden, welche Attribute existieren, Siehe unten
- Anschließend werden auch die Werte geprüft
- Nur was, wenn `attributes` plötzlich ein HTML Element ist?
- Wir müssen also wieder prüfen
  - `if(typeof elem.attributes.item === 'function') …`
  - **Sieht okay aus, oder?**

2. Tja

- Unsere Prüfung war riesiger Mist!
- Es gab einen weiteren Bypass

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

- Jetzt besteht `attributes` aus ***ZWEI*** Elementen
- Und ist somit eine `NodeCollection`
- Die wiederum die Methode `items()` exponiert
- Dammit! Also ein neuer, gründlicherer Check

3. Es geht noch weiter

- Im Verlauf der Tests stellte sich heraus, dass das Iterieren über Attribute schwer ist
- Oft traten Artefakte auf
  - Element hat drei Attribute, zwei wurden entfernt
  - Bei einem ging es gut. Das andere wurde plötzlich unsichtbar. Wurde im Loop nicht gesehen.
  - Das Element wurde zurück ins DOM geschrieben
  - Und schwupps, war das “Geisterattribut” wieder da

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

3. Rückwärts ausparken

- Attribute, so stellte sich heraus, müssen “rückwärts” entfernt werden
- Also von hinten nach vorne. Sonst kommt der Browser nicht hinterher
- Naja, eigentlich ist es ein Indexproblem

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

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

4. Document Clobbering

- Ein weiterer Trick ist der Einsatz von sogenannten “Bösen Bildern”
- DOM Clobbering vom Feinsten

```
<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. Mutationen oder mXSS

- Auch und gerade im DOM ist **mXSS** ein wichtiges Thema
- Browser “mutieren” beim Zugriff auf bestimmte Eigenschaften deren Wert. Einfach so.
  - http://cure53.de/fp170.pdf
- Zum Beispiel `innerHTML` oder `textContent`, `cssText` und diverse andere
- Auch hier gab es eine Menge schmerzhafter Bypasses

```
<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>
```

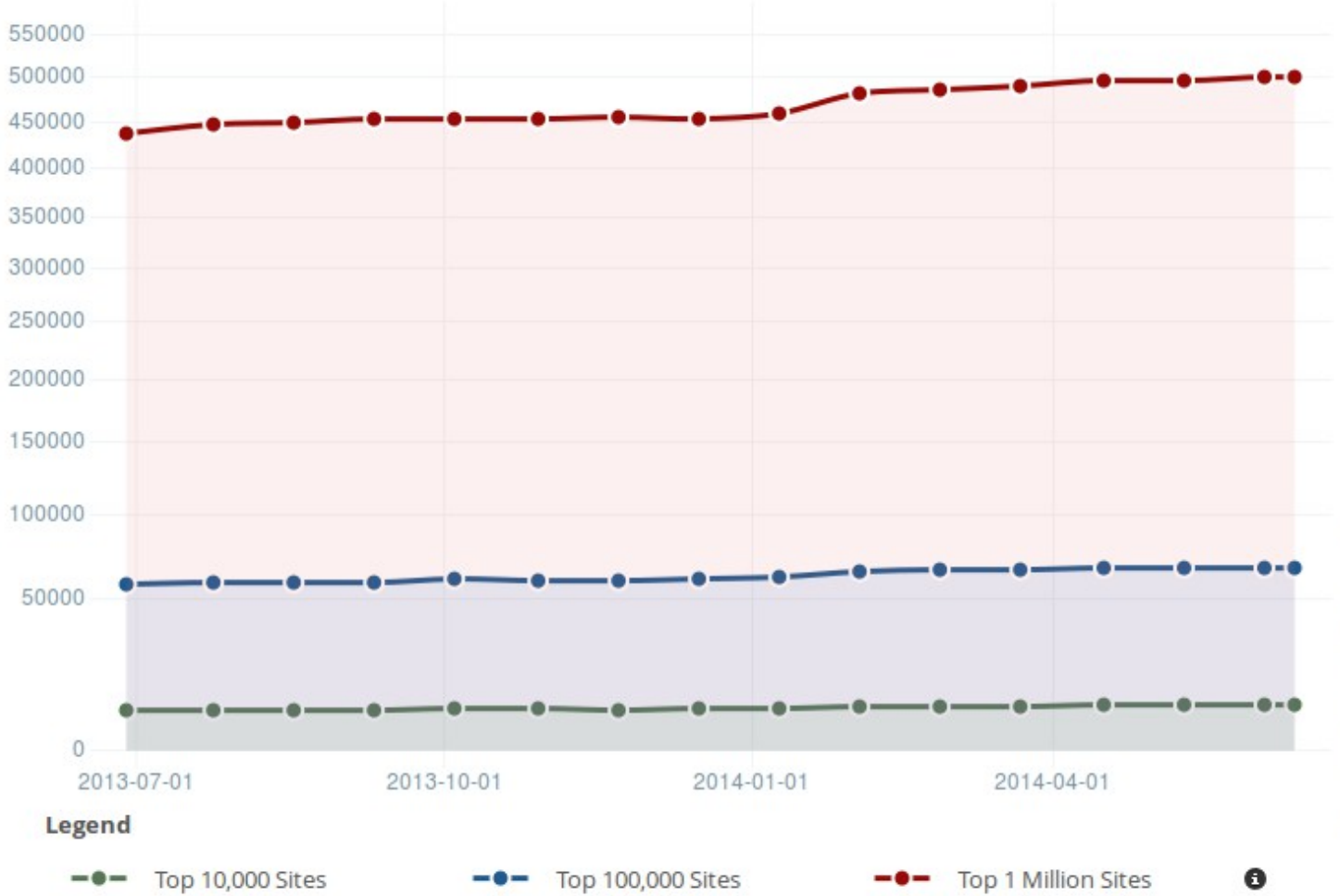
Sicherheit im DOM

- Gibt es nicht. Nur ansatzweise. Schritt für Schritt.
- Folgendes muss beachtet werden
  - DOM Clobbering, Echtheit von Eigenschaften
  - Überschriebene Methoden
  - Mutierende Werte, mXSS
  - Protokoll-Handler mit Unicode
  - Iterieren in richtiger Reihenfolge
  - Verifikation der Änderungen
- **Mit DOMPurifiy sind wir recht weit gekommen**
- Aber sicher sein können wir immer noch nicht
- Und dann gibt es ja noch jQuery und Co.
- Und da geht's erst richtig los. F****g jQuery!!1

jQuery Usage Statistics

Websites using jQuery

Download Lead List



Switch Chart Data

- All Top Site Data
- Top 10k Sites
- Top 100k Sites
- Top Million Sites
- The Entire Internet

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%**

hgi Horst Görtz Institut für IT-Sicherheit

RUB

CURE 53

Fakten

- Die jQuery Library wird ausgesprochen häufig genutzt
  - Gut ein fünftel des WWW laut Statistik. Ein fünftel!
- jQuery hat keine gute “XSS Vergangenheit”
  - Man erinnere sich an `$(location.hash)`
  - Oder `$('<svg onload=alert(1)>')`
  - Die $-Factory, die dynamisch Elemente baut
  - Und diese per `innerHTML` in ein DIV mappt
- Aber es kommt noch schlimmer
- Schauen wir uns folgenden Angriffsvektor an

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

Und nun?

- Der Vektor kann eigentlich kein JavaScript ausführen
- Tut er auch nicht
- Es sei denn, jQuery ist im Spiel – denn jQuery ist “schlau” und wandelt HTML um

```
// 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, "", "" ]
};
```

Also?

- Aus unserem ursprünglich harmlosen HTML String wird nun folgendes

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

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

Und es gibt noch mehr

- Danke, jQuery.
- DOMPurify hat jetzt einen „Safe for jQuery“ Modus
- Ähnliche Dinge lassen sich mit dem Shadow DOM erzeugen
- Das `<template>` Element zum Beispiel
- Obwohl es Kind-Elemente hat, kann man nicht ohne weiteres über diese “drüber-iterieren”

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

<script>
tpl.childNodes // Ist leer, keine Kinder
tpl.content.childNodes // Ah! Da ist unser Element
</script>
```

Selbstschutz

- Was können wir tun, um uns zu schützen?
- Auf dem Server
  - ID und NAME Attribute müssen raus
  - CLASS kann gefährlich werden, wenn MVC Frameworks im Spiel sind
  - Niemals Blacklists bauen, Whitelists sind die einzige Maßnahme
- Auf dem Client
  - Clobbering ist größtes Risiko
  - Ein frisches DOM ist schwer zu bekommen
  - Clobbering sogar in `document.implementation`
- Klassische XSS Lücken verschwinden
- Direkte Angriffe auf das DOM werden interessanter
- Besser jetzt schon Bescheid wissen!

Zusammenfassung

- DOM Security ist schwer
- Das DOM zu verstehen ist nicht immer einfach
- Transaktionen scheitern
- Elemente verschwinden, werden zu neuen Elementen
- Ohne starke JavaScript/DOM Debugger kommt man nicht weit

- Browser kochen nach wie vor ihr eigenes Süppchen
- Erste Schritte sind aber gemacht
  - Dokumentation, Libraries, Browser fixen Standrad-Abweichungen
  - https://github.com/cure53/DOMPurify
- Dennoch, eigentlich bräuchten wir ein Wiki
- Sammlung der verrückten DOM Artefakte
- Und mögliche Sicherheits-Implikationen
- Und Neuheiten kommen jeden Tag hinzu
- Das DOM entwickelt sich rasant(er als alles andere im WWW)

hgi Horst Görtz Institut für IT-Sicherheit

RUB

cure53

Ende

- Fragen?
- Kommentare?
- Vielen Dank!