Dynamic Detection of Object Capability Violations Through Model Checking

Dustin Rhodes
University of California, Santa Cruz
dustin@soe.ucsc.edu

Tim Disney
University of California, Santa Cruz
disnet@soe.ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@soe.ucsc.edu

Abstract

In this paper we present a new tool called DOCaT (Dynamic Object Capability Tracer), a model checker for JavaScript that detects capability leaks in an object capability system. DOCaT includes an editor that highlights the sections of code that can be potentially transferred to untrusted third-party code along with a trace showing how the code could be leaked in an actual execution. This code highlighting provides a simple way of visualizing the references untrusted code potentially has access to and helps programmers to discover if their code is leaking more capabilities then required.

DOCaT is implemented using a combination of source code rewriting (using Sweet.js, a JavaScript macro system), dynamic behavioral intercession (Proxies, introduced in ES6, the most recent version of JavaScript), and model checking. Together these methods are able to locate common ways for untrusted code to elevate its authority.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids—optional subject descriptor

General Terms Languages

Keywords JavaScript; object capability; model checking; proxies; tool;

1. Introduction

The object capability model presents a compelling way to build systems that allow mutually untrusting parties to safely interact by equating the *capability* to accomplish a task with obtaining a reference to a representative object. Object capability systems require support to prevent unintentional reference leaks. Unfortunately, many languages lack the necessary language features to safe-guard references robustly, which makes constructing object capability systems in those languages challenging.

JavaScript would benefit greatly from object capabilities since code written by multiple parties running in the same context is already common (*e.g.* advertisements and mashups). While a common practice, consistently safe-guarding object references in JavaScript is challenging due to the many ways of unintentionally leaking object references. Various tools, language subsets, and libraries [1, 9, 13] have been built to provide the necessary language level features that can enable programming in an object capability world. Even with a sound object capability system in place, it can be difficult for a programmer working in the object capability system to be certain of exactly which capabilities are transferred to a third party.

In this paper, we present a new tool called DOCaT (Dynamic Object Capability Tracer), which can systematically check for leaks in a user's code. DOCaT includes a text editor that highlights the sections of code that can be potentially transferred to untrusted third-party code along with a trace showing how the code could be leaked in an actual execution. This code highlighting provides a simple way of visualizing the references to which untrusted code potentially has access and helps programmers discover if their code is leaking more capabilities than required.

DOCaT is implemented using a combination of source code rewriting (using Sweet.js, a JavaScript macro system) and dynamic behavioral intercession (Proxies, introduced in ES6, the most recent version of JavaScript). Together, these methods are able to locate common ways for untrusted code to elevate its authority. First, plain user code is input into the editor. Then, macros rewrite this code to include a daemon proxy which acts as the untrusted code as well as model checking hooks. Finally, the rewritten code is executed by a model checking algorithm which highlights the leaks it finds.

2. Overview of Object Capabilities in JavaScript

Most websites choose to roll their own security systems and these systems involve some form of static analysis to restrict the user code to a subset of JavaScript. Examples of these systems include Facebook's API [9], Yahoo's AdSafe [1], and Google's Caja [13]. These range from running on a highly restrictive subset of JavaScript, as in AdSafe, to standard JavaScript with some libraries loaded before execution, as in Caja. The overall goal of these is to provide an easy way to run untrusted code in a trusted environment while limiting the power of the untrusted code. All of these systems use some form of object capabilities to accomplish this goal. Caja especially is meant as a system to ease the writing of object capability code in JavaScript.

2.1 Object Capabilities

Object Capabilities limit authority through their use of the reference graph that connects objects. Any object which holds a reference to another object has the power to use that object's capabilities.

A key idea in object capability systems is the principle of least authority (POLA) which means that objects are only given the capabilities they need and no more. In the context of JavaScript a capability is equivalent to a reference to an object or function.

DLS '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-3211-8/14/10. . . $15.00.
http://dx.doi.org/10.1145/020500041.04020408

Figure 1: Object Capability Mint

```
Object.freeze(Array);

function makeMint(name){
  var mint = {
    var purses = [];
    makePurse: function (balance){
      var purse = {
        getBalance: function(){ return balance;},
        sprout: function(){
          return mint.makePurse(0);
        },
        decr: function(amount){
          if(balance-amount<0)return false;
          balance-=amount;
          return true;
        },
        deposit: function(amount,src){
          if(!purses.contains(src))) return;
          if(src.decr(amount)){
            balance+=amount;
          }
        }
      };
      Object.freeze(purse);
      purses.push(purse);
      return purse;
    }
  };
  return mint;
}
```

A JavaScript object capability system allows mutually untrusting modules to interact by exchanging only the capabilities (*i.e.* references) that each module needs to accomplish its task.

For example, in the following code snippet the `untrusted` object needs the `sendEmail` capability so it is passed to the `f` function of `untrusted`.

```
var untrusted = require('untrusted');
var o = {
  launchMissles: function() { /* ... */ },
  sendEmail: function () { /* ... */ }
};
untrusted.f(o.sendEmail);
```

Note that `untrusted` should not have access to the `launchMissles` function since it is not explicitly passed as a reference to `untrusted`. However, in unrestricted versions of JavaScript there are subtle methods of implicitly leaking `launchMissles` that DOCaT can detect and we will describe in the following sections.

A classic example of the power of object capabilities is shown in figure 1. A common use case for such code involves two parties: a bank, who controls the mint, and some users, who have some of the mint's currency. The bank can call `makeMint` which returns a reference to a mint object. This reference provides the bank with capabilities to make purses, which hold currency, and generate new currency. The bank is then free to hand the users references to purses. When writing such a system, it is important that the bank not be required to trust the users. No matter how a user manages their purse reference, the user should not be able to get a reference to a mint or generate new currency.

2.2 ES5 Strict Mode

Writing object capability code in early JavaScript implementations, modeled after EcmaScript 3, is an almost impossible task. Code can easily gain access to the global scope, modify the global objects, and walk the call stack to get references to objects which it should not have. EcmaScript 5, and the advent of `'use strict'` in JavaScript, removes many of these possibilities or provides ways to deactivate them. While many object capability systems choose to restrict the language further, it is possible to write object capability code in pure JavaScript strict mode. DOCaT is built to run on strict mode JavaScript, but no other restrictions to the language are made.

2.3 Object Capability Leaks

Ideally, the programmer of an object capability system should always know what capabilities the system is providing to untrusted code. It is clear that any reference handed directly to the untrusted code is given away, but beyond that it can become difficult to determine who has access to what. This paper presents a tool (DOCaT) to easily check for such leaks automatically. Consider the following examples where important capabilities can become leaked to untrusted code. In each of these examples, the function `untrustedCode` indicates code from an untrusted outside source that should not be able to gain access to the `launchRockets` capability. For simplicity's sake, all code besides the `untrustedCode` function is assumed to be trustworthy.

2.3.1 Leaked Via Global Scope

```
var capability = launchRockets;
untrustedCode();

function untrustedCode(){
  var global = new Function('return this;')();
  steal(global.capability);
}
```

In JavaScript it is possible to get access to the global scope through the `Function` constructor and `eval` as seen in the function `untrustedCode`. By gaining access to the global scope the untrusted code can steal the `launchRockets` capability. To fix this problem, the programmer needs to overwrite the possible methods of obtaining global scope, `eval` and the `Function` constructor, or hide their secret inside of a closure.

2.3.2 Leaked Via Return Value

```
function f(){ return launchRockets; }
untrustedCode(f);
```

Untrusted code can call functions that are provided to it and gain access to their return values.

2.3.3 Leaked Via Poisoning Globals

```
function hide(){
  console.log(launchRockets);
}
untrustedCode(hide);
```

Untrusted code can overwrite globals so that they function as normal, but also steal references. In this case, console.log can be overwritten by the untrusted code to steal all references passed to it. Freezing an object prevents any of its members from being modified. Thus, this attack can be prevented by freezing globals before using untrusted code.

2.3.4 Leaked Via This Passing

```
var obj = {
    locked:true,
    f:function(){
        if(!this.locked)return launchRockets;
        else return null;
    }
};
Object.freeze(obj);
untrustedCode(obj);
```

```
function untrustedCode(obj){
  var login = obj.f.apply({locked:false},[]);
  leak(login);
}
```

JavaScript's `this` binding functions differently than in most classical object oriented languages. It is possible to pass in an arbitrary `this` object when making a function call by using `Function.apply`. This possibility means that checks on `this`, as above, which look safe are potentially dangerous.

2.3.5 Leaked Via Proxy Setters

```
var obj = {
  f:function(){
    this.temp = launchRockets;
    //... code which uses temp
    this.temp = 0;
  }
};
untrustedCode(obj);
```

Here, it looks as though `f` never finishes with any sensitive information on the `this` object, so nothing is leaked. However, by using proxies, control can switch to the untrusted code during the assignment at which point `launchRockets` is stolen.

2.3.6 Leaked Via Shallow Freeze

```
var a = {
  link:b
};
var b = {
  f:function(x){
    console.log(x);
  }
};
Object.freeze(a);
untrustedCode(a);
var secret = launchRockets;
b.f(secret);
```

`Object.freeze` is only a shallow freeze and does not freeze all objects which are transitively accessible from the frozen object. Therefore, even if all objects are frozen before passing them to untrusted code, the untrusted code can still gain read/write access to other references through them. This leak is fixed by calling `Object.freeze` on all accessible objects or by using a deep freeze function.

2.3.7 Leaked Via Deep Freeze Return

```
var a = {link:function(){ return b; }};
var b = {f:function(x){console.log(x);}};
DeepFreeze(a);
untrustedCode(a);
b.f(launchRockets);
```

Even performing a deep freeze before handing references to untrusted code is not completely safe from leaking. A deep freeze will not freeze the results returned by functions. Thus, just as last time, `b` does not end up frozen, but the untrusted code can gain access to it through `a`.

3. Overview of JavaScript Proxies

Proxies [5] are a key aspect of our implementation of DOCaT, so we briefly review JavaScript's proxy interface in this section. Proxies [17] [6] are a relatively recent addition to JavaScript, which allow code to define its own custom behavior for objects. Proxies define custom behavior by converting operations on the object into function calls called traps.

The JavaScript proxy constructor takes a prototype and a handler for the proxy. The handler is an object which defines the proxies' traps. One of the most basic pair of traps are the get and set traps, which define the proxies behavior when it is used as a record. For example, the get trap in figure 2 will cause the proxy to always return 42 in response to a property access.

Figure 2: Simple Proxy

```
var handler = {
  get: function(target, name){
    return 42;
  }
  set: function(target, name, value){
    target[name]=value+100;
    return true;
  }
  apply: function(target, thisValue, args){
    return args.length;
  }
}

var proxy = new Proxy(Function,handler);
console.log(proxy.x);//prints 42
console.log(proxy['y']);//prints 42

proxy.x = 1; //proxy.x is now 101

console.log(proxy('a',2,'b'));//prints 3
```

In the get trap, target is the unproxied prototype object and name is the name of the property being requested. When `proxy.x` is evaluated, get is called with target equal to `Object` and name equal to the string `'x'`. By returning 42 in our get function, any property accessed on our proxy will return 42.

Set and apply, the other two traps implemented in figure 2, follow the basic form of get but add some complexity. Set has target and name which work exactly like get, but adds a value parameter which is the right-hand side of the assignment operator. The set trap in figure 2 traps all set commands and sets the property to the value plus 100.

The last trap in figure 2 is the application trap. It defines behavior for when the proxy is used as a function, as opposed to a record including cases of `proxy()` and also `proxy.apply(this,args)`. The application trap takes three parameters: the target object, the binding for `this`, and the arguments array. In the example proxy, the application trap just returns the number of arguments it was passed.

4. Implementation of DOCaT

DOCaT is implemented using three complementary parts. The first part is a daemon proxy which acts as a stand-in for an untrusted source in the object capability code. Second, there is a macro system in place which is used to add additional hooks to the source code which in turn are used by the daemon proxy and model checker. Finally, there is a top level model checker which exhaustively searches the code for leaks using the daemon proxy and macro hooks.

4.1 The Daemon Proxy

The daemon proxy, shown in figure 3, is designed to act as an adversarial untrusted source in object capability code and, as such, tries to gain a reference to as many capabilities as possible. In all

Figure 3: Daemon Proxy

```
let daemonHandler = {
    // Traps *.x and always returns daemon.
    get: function (target, name) {
        return daemon;
    },
    // Traps *.x=y and leaks y.
    set: function (target, name, val) {
        explore(val);
        return true;
    },
    // Traps *(x) and x.*(y).  X and y are both leaked.
    apply: function (target, thisValue, args) {
        explore(args);
        explore(thisValue);
        return daemon;
    }
};

// First create the daemon proxy.  The prototype is Function so it can be called.
var daemon = new Proxy(Function,daemonHandler);
//Keep a list of all explored references.
var explored = new WeakMap();

// Explore does a deep search of target, leaking everything it finds.
function explore(target) {
  // If the target is daemon or has already been explored, then skip it.
  if(target===daemon || explored.has(target))return;

  // Put the target in the explored map.
  explored.set(target,true);

  // If target is an object, do a deep search and set all its properties to daemon.
  if (typeof target == 'object') {
    for (var key in target) {
      explore(target[key]);
      target[key] = daemon;
    }
  }
  // If target is a function, call the function passing in daemon as this and all parameters.
  if (typeof target === 'function') {
    explore(target.apply(daemon,[daemon]));
  }
  // Use the map created by wrap (fig. 4) to highlight the leaked lines.
  editor.highlight(allocLine.get(target));
}
```

code run through DOCaT, code coming from an untrusted source is replaced with an instance of the daemon proxy. The key idea of this daemon proxy is that whenever the daemon gains access to an object o, it recursively explores all objects transitively accessible from o, as well as marks the source code where those values were created as leaked. The daemon acquires these references by implementing traps for get, set, and apply.

The get trap is very straightforward. No matter which property one attempts to get from the daemon, it always returns a daemon. Any property coming from an untrusted source should also be untrusted. The get trap works for chained values as well, for instance `daemon.x.y` evaluates to daemon because `daemon.x.y` $\rightarrow$ `daemon.y` $\rightarrow$ `daemon`.

If a property is set on the daemon, via the set trap, then the function `explore` is called to identify new capabilities transitively accessible through the assigned value. If the value being explored is an object, then the daemon recursively calls explore on each of its properties after which it sets that property equal to daemon. To explore a function the daemon passes a daemon for the `this` value of the function as well as all parameters to the function. In this way, if the function might leak information to `this` or any of its parameters, the daemon will gain access to them. Any value returned by the function is explored as well.

Finally, there is the possibility that someone will call the daemon as a function or method and pass in parameters to it. In this case, the apply trap does a combination of the get and set behavior. It explores the value `this` is bound to and also all the arguments that were passed in, and finally it returns a daemon.

4.2 Proxy Example

To understand how these traps gain access to as many references as possible, it can be helpful to see a small example in action. The following is a simple snippet of source code where the untrusted

Figure 4: Binary Operation Trap

```
//JavaScript proxies can not trap such operations
//so we trap all binary and unary operations
//with sweet.js macros as such.
operator + 12 left {
  { $lhs, $rhs } =>
  { add($lhs,$rhs); }
}
//If either side is a daemon return daemon.
//Otherwise, return the result of the operation.
function add(lhs,rhs){
  if(lhs===daemon || rhs===daemon)
    return daemon;
  else return lhs+rhs;
}
```

source has been replaced with a reference to the daemon:

```
1 var x = 'secret1';
2 var y = 'secret2';
3 var obj = {
4   prop:x,
5   fun:function(arg){ arg.a = y; }
6 };
7 daemon.fun(obj);
```

The execution of this code would look like this
$daemon.fun(obj)$
→ (daemonHandler.get(daemon, 'fun'))(obj)
→ daemon(obj)
→ daemonHandler.apply(daemon, daemon, obj)
→ explore(obj)
→ explore(obj.prop); explore(obj.fun)
At this point 'secret1' has leaked.
→ explore(obj.fun)
→ explore(obj.fun.apply(daemon, [daemon]))
→ explore(daemon.a = y)
→ explore(daemonHandler.set(daemon, 'a', y))
→ explore(y)
At this point 'secret2' has leaked and execution has finished. Furthermore, all the properties of obj are now set to daemon, potentially causing more leaks if they continued to be used in other parts of the program.

4.3 Trapping Operators

Using proxies, the daemon is able to trap all incoming get, set, and apply operations but can not trap binary and unary operations such as $+$, $-$, or $*$. JavaScript proxies do not offer a way to trap these operations, so they are instead trapped using macros. These macros are used to rewrite user code in such a way that the binary and unary operations can be trapped.

Sweet.js [21] is a macro system written for and in JavaScript. It is designed to take standard JavaScript code plus macros and expand the macros within to yield pure JavaScript. Using these macros, it is possible to alter the behavior of operators. The macro in figure 4 changes all occurrences of the symbol + into calls to the function add. For example, it expands the code `var z = x + y;` into the code `var z = add(x, y);`. The function `add` checks if either side is a daemon and if so returns a daemon. Otherwise, it returns the standard result of the operation. All other unary and binary operators are handled in a similar fashion.

Figure 5: Branch Transform

```
//trap if calls so we can model check

macro if {
  rule{ ($test:expr) { $then ... } }
  => {
    if(branch($test)) { $then ... }
  }
  //Trap all other if cases.
  //...
}
```

4.4 Model Checker Algorithm

The overall goal of DOCaT is to attempt to find all possible leaks to an untrusted source in the user's code. If the source code had no branches, the daemon proxy and macros would be enough to find all these leaks on their own, however, if the source code performs a conditional test on a value provided by the untrusted source, both paths will need to be tested for leaks. To do this, a model checking algorithm is used to exhaustively check all paths which the untrusted source has control over for leaks. The sweet.js macro shown in figure 5 introduces a call to the `branch` function whenever there is a conditional test in the target program. Using these `branch` calls, it is possible to check all paths of the target code using a model checking algorithm [4].

In order to run its algorithm, the model checker first appends the macros to the user's code where the untrusted source has been replaced by a daemon reference. The model checker then calls `sweet.compile` as seen in line 9 of figure 6. This function returns expanded code which is pure JavaScript as seen in figure 9. After the code is expanded, it can then be checked for leaks.

To see how the model checking algorithm works, consider this example code with `branch` calls inserted:

```
1 var d = daemon;
2 if(branch(d.x)){
3   if(branch(d.y)){
4     d.x='leak1';
5   }else{
6     d.y='leak2';
7   }
8 }
```

In order to properly check this code for leaks, there are three paths which must be evaluated, `[true, true]`, `[true, false]`, and `[false]`. These three paths are executed by the main loop inside of the `testCode` function, shown in figure 6. The model checker uses two variables to keep track of its execution of the various paths. The `path` variable is a list of booleans, which shows the decision made at each branch point in the current path. This list starts out empty and gradually fills up as code is evaluated. `ifNumber` holds the current position in the path. This number is reset to zero before each evaluation and will finish each evaluation as being equal to the length of `path`. To evaluate one path, the algorithm resets `ifNumber`, evaluates the expanded code, then adjusts the path variable to take a new path for the next evaluation.

The first time through the loop, `path` will be an empty array and `ifNumber` will be 0. At this point, when the model checker reaches the first call to `branch`, it must decide which path to take. In this algorithm, the model checker always explores true paths first, followed by the false path. The first time reaching a particular `branch`, `true` is returned and added to the path. In this example, the first time hitting the `branch` on line 2 and 3 will return `true` for this reason. These two returns will cause the model checker to explore the first path, which is `[true, true]`. At this

Figure 6: Model Checking

```
// Get the user's code and append our macros to its beginning,
// then macro expand the user's code using sweetjs.
// The proxy handler and all its code is not macro expanded.

// Path and ifNumber are used by the model checker.
var path = [];
var ifNumber = 0;// remember which branch we are on
function testCode(userCode){
  var expandedCode = sweet.compile(macros+userCode);
  //the branches used in model checking
  do{
    ifNumber = 0;
    // run the user's code, which will fill up the path array
    // on branches.
    eval(expandedCode);
    // Now that path is full pop off all false paths (we
    // have already gone down the true and false path for them) and
    // change the last true to be false.
    while(path[path.length-1]==false && path.length>0){
      path.pop();
    }
    if(path.length>0) path[path.length-1]=false;
  }while(path.length>0);
  // If path is empty we've explored all paths so
  // we are done.
}


function branch(test){
  if(test!==daemon) return test;
  // Do model checking.
  ifNumber++;
  // The first time hitting a branch pick true.
  if(ifNumber>path.length){
    path.push(true);
  }
  // Return the choice our model checker tells us to.
  return path[ifNumber-1];
}
```

point, `'leak1'` is leaked and the evaluation terminates. DOCaT has gained some knowledge on what is leaked, but the model checker is not done yet as it has two more paths to evaluate.

After one evaluation of the user code, a new path is generated by flipping the last choice to a false. If the last choice is already false, then it is popped off and the new last choice is considered. With a new path generated, the user code must be evaluated again. In this way, after exploring the `[true, true]` path, the algorithm moves onto the `[true, false]` path and finally the `[false]` path. At this point, all paths in the user code have been explored and any reference that leaked on any path is highlighted.

4.5 Allocating Line Numbers

Having found a leak, it is important to correctly display the leak. To show which references have leaked, DOCaT highlights the line number in which the leaked reference was created. Finding these line numbers is accomplished using the macro shown in figure 7. While the code may seem confusing, its purpose is simple. Since the values that are leaked are always created with an assignment operator, the macro also runs on an assignment operator.

Every time the macro sees an assignment operator, the macro grabs the equals symbol and whatever occurs after it, up until a semicolon. To get an integer for the line number, the macro uses the line number on which the assignment operator is located. The macro then takes everything that occurred to the right of the equals sign and passes it into a function, called `wrap`, along with the line number. `Wrap` creates a weak map associating objects to line numbers. When a new call to `wrap` is made, it adds the object to the map with the associated line number and returns the object.

Now, if that object is ever leaked somewhere later in code, the daemon can tell which line number it was initially assigned on by looking it up in `wrap`'s table. This allows the daemon to highlight the correct line number in the editor. To illustrate these various macro transformations, figure 8 shows some JavaScript source code, with the expanded code in figure 9 including the calls to `wrap`, `add`, and `branch`. These figures make it clear how the macros alter the input code.

4.6 Additional Complexities

JavaScript is a large and complicated language, and while the implementation described above catches most errors, there are a couple other edge cases which are implemented in the working version but are not shown in the simplified sample code. One such edge case is the possibility for an untrusted source to get access to the global scope using the `Function` constructor or `eval.call` in JavaScript [22].

Figure 7: Variable Wrapping

```
// Whenever there is an assignment, we record
// the line number so if the object that was
// assigned is leaked we can highlight its
// original location.
let (=) = macro{
  case { $assignSym $after ...; } =>
  {
    // Get the line number of the = sign.
    var line =
      (#{$assignSym}[0].token.sm_lineNumber);
    // Create a token representing
    // the line number.
    letstx $number = [makeValue(line,
                                #{$assignSym})];
    return #{= wrap($after ...,$number)};
  }
}$

// A weak map holds the line numbers for
// all objects.
var allocLine = new WeakMap();
function wrap(obj, line){
  // Handle primitive cases not shown.
  allocLine.set(obj,line);
  return obj;
}
```

Figure 8: Input Code

```
var x = 'secret';
var y = x.length() + 3;
if(y){
  console.log(y);
}
```

Figure 9: Compiled Code

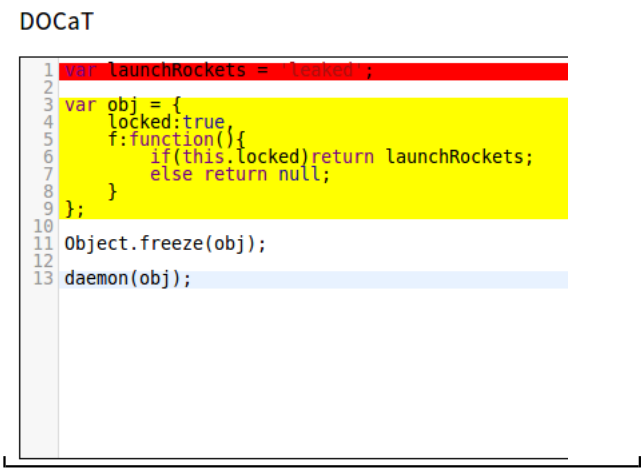
```
var x = wrap('secret',1);
var y = wrap(add(x.length(), 3), 2);
if(branch(y)){
  console.log(y);
}
```

To simulate an untrusted source getting access to global, as soon as the first daemon object is loaded into user code, it tries to grab the global scope using both the `Function` constructor method and the `eval` method. If either of these work, it passes the global scope to `explore`. This attack can be prevented, as is done in CAJA, by overwriting the Function constructor (along with the ways of getting a hold of it such as the prototype Function's constructor value) and `eval.call` method to a safe alternative before loading unsafe code.

A similar attack can be performed on JavaScript's built-in global variables. An untrusted source can attempt to overwrite commonly used built-in globals with its own adversarial versions as soon as it gains control [11]. Again, the first time a daemon is loaded, it attempts to overwrite all globals while preserving their functionality.

JavaScript provides a way to prevent the overriding of object properties through freezing the object. An object is frozen by passing the object to be frozen to the built in method `Object.freeze`. A frozen object can have its properties viewed but not modified. By freezing the global objects before the untrusted source has a chance to edit them, it is possible to prevent the untrusted source from overwriting the built-in globals.

Figure 10: DOCaT Screen Shot



Finally, DOCaT must take into account that some variables which are leaked are already frozen. If the daemon receives a frozen object, attempting to overwrite its properties with daemons will fail. The daemon can, however, continue to explore the object. In the daemon's highlighting, it uses two colors: red to represent something which is leaked and unfrozen, offering read and write access, and yellow to represent leaked frozen objects which only provide read access.

5. Validation

5.1 Examples

Now that the inner workings of DOCaT have been explained, it is helpful to see specific examples of its use. It is possible to test all the examples seen in the introduction of this paper using this tool by simply replacing instances of `untrustedCode` with `daemon` and running the code through DOCaT. To illustrate how the tool works, recall this example from the introduction.

5.1.1 Leaked Via This Passing

```
untrustedCode = daemon;
var obj = {
    locked:true,
    f:function(){
        if(!this.locked)return launchRockets;
        else return null;
    }
};
Object.freeze(obj);
untrustedCode(obj);
```

When the daemon is handed `obj`, it will call the function `f` during its exploration of `obj`. When it calls `f`, it will pass in `daemon` for `this` as well as for all parameters. Thus, inside the function `if(!this.locked)` will evaluate to `if(!this.daemon)` which will evaluate down to just `if(daemon)`. This branch is on a `daemon` so both paths will be executed, the true path leaking `launchRockets`. A screen shot of this code being analysed by DOCaT is shown in figure 10.

5.1.2 Mint

A mint, shown in the introduction in figure 1, is often used as an example case of the power of object capabilities. `MakeMint` is used to generate mint references, presumably for a bank. Once the bank has a mint reference, the bank can make purses for that mint

and create currency. If a user has a purse, the user can also make more purses, but the user can not create currency. If the user has two purses, they can deposit money from one to the other, but the user should not be able to create new money. The mint shown in figure 1 is a working version of a mint without leaks in JavaScript.

Figure 12 shows some of the possible errors a programmer could make when writing their mint. These errors mean a user could gain capabilities that only the bank should have. The first error simply removes the freezing of Array. If this line is removed, the user can edit `Array.push` which causes a leak on line 23. This edit leads to leaking the purse's array to the user, as well as any purse later created by the mint.

For the second example, a single-letter change freezes the entire array of purses instead of the individual purse that is returned by `makePurse`. However, without the freeze, the user can modify the properties of the purse including the `purse.decr` method. The fact that `decr` is controlled by the user means that the call to `decr` on line 17 is highlighted as an unsafe branch. Since this branch protects the balance adjusting code, an unsafe branch here means the user will be able to create money.

Finally, consider a different strategy for verifying that two purses belong to the same mint. It would be natural, to add a `checkMint` function which takes a mint and verifies that its mint matches the incoming mint. Making this change seems safe, as `purse` is frozen so the user can not tamper with its functions to change `checkMint` into something dangerous. However, the user may not find such tampering necessary, as they can simply pass in a fake object which has an unsafe `checkMint` function. Then, when `checkMint` is called on this fake object, the user will be handed a reference to the purse's mint, granting them all the capabilities of the bank.

5.1.3 Membranes

Membranes are often used when writing object capability code [16]. In object capability code, it is important to control who has access to which references. Simply handing over a reference to another source often provides too much control to that source. Thus, it is common to place a membrane around the reference before it is passed to untrusted code. Wrapping a reference in this way allows the owner certain abilities, such as the ability to revoke the reference later, as the membrane shown in figure 11 does. When writing a membrane, it is important that the untrusted source can only access references which are wrapped in a membrane.

Even when writing a simple membrane such as the one in figure 11, there are some complexities that might not be immediately obvious. For example, it might not seem necessary to write a set trap. It is quite obvious that any references coming out of the membrane will need to be wrapped, but it is less clear that references going in must be as well. By itself, an untrusted source putting references into the wrapped object is unable steal any references, but as soon as the owner of the object tries to use it, it is possible that the untrusted code could run and pass the untrusted source an unwrapped reference. This attack is prevented by wrapping all incoming references in the set trap, so if these references are ever run, they will be wrapped by the membrane.

It also might seem like there is no need to wrap the incoming `this` argument on an apply trap. If functions could only be called using dot syntax, the `this` argument would always be the expected object, so there would be no need to wrap it. However, if `this` is not wrapped, the untrusted code can call the function using `Function.apply` and pass in an arbitrary `this` object. If the `this` object is used to store references, these references could be assigned to the passed in `this` object which could report them back to the untrusted code, all without ever passing through a membrane.

Leaks such as these can be found using DOCaT. So far, there has only ever been one source of unknown actions. Here, there are two: the code which has a reference wrapped in the membrane, and the untrusted code which will be passed this membrane. The important feature of a membrane is that these two sources should not touch. Here two daemons represent these two sources: `daemon1` represents the untrusted source that the membrane will be handed to and `daemon2` represents the user code which the membrane will be wrapping. The membrane code can then be tested with the code `daemon1(makeMembrane(daemon2))`.

Figure 11: Revocable Membrane

```
1  function makeMembrane(initTarget){
2    var enabled = true;
3
4    var handler = {
5      get: function (target, name){
6        if(!enabled)throw new Error('revoked');
7        return wrap(target[name]);
8      },
9      set: function (target, name, val){
10       if(!enabled)throw new Error('revoked');
11       target[name] = wrap(val);
12       return true;
13     },
14     apply: function (target, thisValue, args){
15       if(!enabled)throw new Error('revoked');
16       var result = target.apply(wrap(thisValue),
17                                 args.map(wrap));
18       return wrap(result);
19     }
20   };
21
22   function wrap(target){
23     return new Proxy(target,handler);
24   }
25
26   return {
27     wrapper: wrap(initTarget),
28     revoke: function(){ enabled = false;}
29   }
30 }
```

5.2 Performance

DOCaT runs well on all of the examples given in this paper and checks them all for leaks in under 200 milliseconds each. There are three performance considerations to keep in mind when running DOCaT. First, there is a compilation step where the sweet.js macros are appended to the input code and then compiled using sweet.js.

The second issue is the added complexity of the compiled code in comparison to the source code. This complexity amounts to binary operations, unary operations, and branches being replaced with function calls, some objects being replaced by the daemon proxy, and the time it takes to explore the leaked reference graph. The replacement of simple operations with function calls adds a linear amount of time to program execution. The daemon proxies add to execution time because the proxy API is relatively new and therefore has not been heavily optimized. As the proxy API ages, the daemon proxy speed should rise. The time it takes to explore the reference graph depends heavily on how much is leaked to the daemon. The principle of least authority [20] states that as few references as possible should be given to an untrusted source so, in most cases, the number of references leaked to the daemon will be relatively small compared to the total number of references.

Finally, there is the model checking portion of DOCaT. The

Figure 12: Examples

Example	Patch	Leaked Lines	Error Caught
Mint	1:/ Object.freeze(Array)/	Line 5: purses Line 7: purse	✓
Mint	22: `Object.freeze(purses);`	Line 7: purse Line 17: branch	✓

Mint	5: // var purses = [];
	16: if(src.checkMint(mint))
	
	7:
	`var purse = {`
	`checkMint: function(otherMint){`
	`if(mint===otherMint)return true;`
	`else return false;`
	`}`
		Line 4: mint	✓

Membrane	9: // set: function(target,thisValue,args)	Line 30: obj	✓
Membrane	16: var result = target.apply(thisValue...	Line 30: obj	✓
2.3.1 Leaked Via Global Scope	as is	launchRockets	✓
2.3.2 Leaked Via Return Value	as is	launchRockets	✓
2.3.3 Leaked Via Poisoning Globals	as is	launchRockets	✓
2.3.4 Leaked Via This Passing	as is	launchRockets	✓
2.3.5 Leaked Via Proxy Setters	as is	launchRockets	✓
2.3.6 Leaked Via Shallow Freeze	as is	launchRockets	✓
2.3.7 Leaked Via Deep Freeze Return	as is	launchRockets	✓

model checking adds significant time as the entire compiled code needs to be executed multiple times depending on the number of untrusted branches. In the worst case, the model checking can increase evaluation time exponentially. However, because object capability code usually attempts to limit its use of untrusted code there should not be a large number of branches on untrusted objects, therefore the exponential explosion, which is theoretically possible, should not happen in practice.

5.3 Limitations

While DOCaT can catch most leaks that may happen in a program, there are a few that slip by it. Most of these leaks theoretically can be caught, but would require significantly more complexity in the model checker and would have a prohibitively long run time. Finding ways to address these problems without increasing run time beyond a useful length or introducing too many false positives is an important area for future research on this subject. The simplest leaks that slip by DOCaT are a classic problem in model checking, looping on an unknown value. The model checker can pick an arbitrary number of times to run the loop, but there could always be a leak that happens only on the next loop iteration. If we actually wanted to fix this problem, the model checker would need to label all variables that change inside of an adversarial controlled loop as adversarial themselves. The model checker can not determine the number of times the loop runs, so it can not know the results of these variables. By propagating these new unknowns through the program, the model checker could still catch all leaks due to loops, but would introduce false positives.

Similar to this problem, but slightly more subtle, is the problem of what order to execute leaked functions in. Theoretically, the order the daemon executes them in could cause different leaks to occur. To solve this problem, the daemon would ideally have to execute all leaked functions in all possible orders. This kind of execution takes drastically too much time and in practice, these leaks do not occur with much frequency. Alternatively, the problem could also be addressed by taking into account which variables the daemon has indirect control over and assuming these as adversarial unknowns. This would require a more complicated macro system as well as introduce some more false positives.

Finally, there is the issue of functions which behave differently depending on the number of parameters passed in. A function could check to see if the 100th argument was undefined and, if it was not, then leak. The daemon can see how many arguments a function expects by looking at its length property, however, there is no way to know if it will act differently if the daemon passes in a different number of parameters. One could eliminate this problem by forbidding use of the arguments array, but running on standard JavaScript has been determined to hold greater priority.

These leaks prevent us from being able to guarantee that all possible leaks will be caught by DOCaT. Fixing these leaks, while keeping execution at a reasonable speed and not introducing too many false positives, is important for future research.

6. Related Work

This work draws greatly on proxies, model checking, and object capabilities. The JavaScript proxy work done by Miller and Van Cutsem [17] [6], as well as by T.H. Austin. [2], provide the basis for the JavaScript proxies used by this paper. Proxies such as these have been implemented in various other languages, such as Racket's surrogates [10] and Smalltalk [12] and Python's [24] doesNotUnderstand method as well as the language E [19] which is also written with object capabilities in mind.

The use of proxies in model checking algorithms is previously done by Alessandro Bruni [4]. Other abstract execution papers also aided in the creation of the model checking algorithm. [23]

Mark Miller's work in his thesis [16], papers [18], and Google's Caja [13] are an important foundation for work on Object Capabilities in JavaScript. Sophia Drossopoulou and James Noble [8] outline why there is a need for object capabilities in the web world and also, some of the current difficulties with writing object capability code. Maffeis' paper [15], also helps to define why object capabilities are a good choice for interaction over the internet. Google's Caja [13] displays the difficulty of writing object capability code in EcmaScript 3, but the viability of writing it in EcmaScript 5.

Adam Barth [3] and Vineeth Kashyap [14] both deal with Object Capability leaks in JavaScript code but do so from a browser perspective and guard against cross-origin JavaScript capability leaks specifically. Christos Dimoulas [7] describes a different way of verifying object capability code using contracts and information flow theory to check capabilities at run time.

7. Conclusion

Object Capability code provides a simple framework for interaction between trusted and untrusted code. While the framework is simple, the possibility for untrusted code to act as a malicious attacker means that object capability code must strive to eliminate any means by which the untrusted code can upgrade its authority. In an object capability system, upgrading authority means obtaining references to new capabilities. DOCaT provides a simple way that such leaks could be spotted in object capability code before deployment.

8. Bibliography

References

[1] Adsafe. `http://www.adsafe.org/`, accessed June 2014.

[2] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. *SIGPLAN Not.*, 46(10):921–938, Oct. 2011.

[3] A. Barth, U. Berkeley, J. Weinberger, and D. Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. *In Proc. of the 18th USENIX Security Symposium (USENIX Security 2009)*, 2009.

[4] A. Bruni, T. Disney, and C. Flanagan. A peer architecture for lightweight symbolic execution. 2013.

[5] T. V. Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP 2013*, 2013.

[6] T. V. Cutsem and S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Dynamic Languages Symposium*, 2010.

[7] C. Dimoulas, S. D. Moore, A. Askarov, and S. N. Chong. Declarative policies for capability control. Institute of Electrical and Electronics Engineers, 2014.

[8] S. Drossopoulou and J. Noble. The need for capability policies. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 6:1–6:7, New York, NY, USA, 2013. ACM.

[9] FacebookAPI. `https://developers.facebook.com/docs/reference/php/facebook-api/`, accessed June 2014.

[10] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `http://racket\-lang.org/tr1/`.

[11] Global object poisoning. `http://code.google.com/p/google-caja/wiki/GlobalObjectPoisoning`, accessed June 2014.

[12] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[13] Caja. `http://code.google.com/p/google-caja/`, accessed June 2014.

[14] V. Kashyap and B. Hardekopf. Security signature inference for javascript-based browser addons. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 219:219–219:229, New York, NY, USA, 2014. ACM.

[15] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. *Dep. of Computing, Imperial College London, Technical Report DTR10-04*, 2010.

[16] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[17] M. S. Miller and T. V. Cutsem. Catch-all proxies. `http://wiki.ecmascript.org/doku.php?id=harmony:proxies`.

[18] M. S. Miller, T. V. Cutsem, and B. Tulloh. Distributed electronic rights in javascript. *ESOP'13 22nd European Symposium on Programming*, 2013.

[19] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[20] M. S. Miller, B. Tulloh, and J. S. Shapiro. The structure of authority: Why security is not a separable concern. In *Proceedings of the Second International Conference on Multiparadigm Programming in Mozart/Oz*, MOZ'04, pages 2–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[21] Sweet.js. `http://http://sweetjs.org/`, accessed June 2014.

[22] Global scope reachable via this. `http://code.google.com/p/google-caja/wiki/GlobalScopeViaThis`, accessed June 2014.

[23] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. *SIGPLAN Not.*, 47(10):537–554, Oct. 2012.

[24] G. van Rossum and F. Drake. *Python Reference Manual*. PythonLabs, Virginia, USA, 2001.