6.858 Fall 2014 Lab 6: Javascript isolation

Handed out: Lecture 16
Due: Two days after Lecture 19 (5:00pm)

Introduction

In this lab, you will implement a system to allow a limited set of Javascript to execute as part of zoobar user profiles. You will implement a combination of static rewriting and dynamic sandboxing to ensure that code running as part of the profile cannot modify the rest of the page, but yet it can make some changes to HTML elements that were part of the profile itself.

To give you an example of the kind of profile code that we will support, a user should be able to place the following code in their zoobar profile:

```
<div id="a">x</div>
<div id="b">x</div>
<div id="c">scrolling message.. </div>
<div id="count"></div>
<script>
    var count = 0;

    function flip(a, b) {
        document.getElementById(a).textContent = "nothing here";
        document.getElementById(b).textContent = "-- click me! --";
        var bump = function (x) { return x+1; }
        count = bump(count);
        document.getElementById('count').textContent = 'click count: ' + count;
    }

    flip('a', 'b');
    document.getElementById('a').onclick = function() { flip('a', 'b'); };
    document.getElementById('b').onclick = function() { flip('b', 'a'); };

    function scroll(id) {
        var s = document.getElementById(id).textContent;
        var ns = s.substring(1) + s[0];
        document.getElementById(id).textContent = ns;
        setTimeout(function() { scroll(id); }, 100);
    }

    scroll('c');
</script>
```

and get a profile that looks like the following:

```
nothing here
-- click me! --
message.. scrolling
click count: 1
```

You will build an HTML/Javascript rewriter that will ensure that this code cannot tamper with the rest of the page, steal the cookies, etc.

The system you will be building will be a simpler version of Facebook's original FBJS system. You may find it useful to refer to the paper on Run-Time Enforcement of Secure JavaScript Subsets to understand how it works. Note that Javascript isolation in general is a very difficult problem, and most systems that have been developed have historically turned out to be insecure in a variety of ways. Although we are not aware of any vulnerabilities in the system that you will be building in this lab assignment, it has not been thoroughly vetted or analyzed, and could very well have some subtle holes in it. (If you find any, let us know!)

First, log in as the `httpd` user, check in your solution for lab 5, and fetch the new code for lab 6. For those using the provided .zip files, please download lab6.zip from the MIT OpenCourseWare site. Note that, for simplicity, you do not need to integrate changes from previous labs into this lab; we will focus just on rewriting HTML code in profiles for now.

```
httpd@vm-6858:~$ cd lab
```

```
httpd@vm-6858:~/lab$ git add answer-1.txt answer-2.html answer-3.html answer-4.txt answer-chal.html
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab5'
[lab5 dc6f228] my solution to lab5
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
Already up-to-date.
httpd@vm-6858:~/lab$ git checkout -b lab6 origin/lab6
Branch lab6 set up to track remote branch lab6 from origin.
Switched to a new branch 'lab6'
httpd@vm-6858:~/lab$
```

Now, build and run this code as before:

```
httpd@vm-6858:~/lab$ make clean
rm -f *.o *.pyc *.bin zookld zookfs zookd zooksvc *.log
httpd@vm-6858:~/lab$ make
...
httpd@vm-6858:~/lab$ ./zookld
...
```

Javascript rewriting

To understand how we will isolate Javascript code, let's first examine the new code in this lab. We have implemented a new function, called `filter_html`, in `zoobar/htmlfilter.py`, which sanitizes user profiles. This function is invoked from `users.py` on each user profile. The `filter_html` function does three things, as follows.

- First, it parses the HTML using the lxml library, and strips away any dangerous tags and attributes (such as `<style>` tags). It also rewrites the `id` attributes on all HTML elements by prepending the string `sandbox-` to them. This will help us later identify at runtime which DOM elements belong to the profile, and which DOM elements do not.
- Second, this function finds `<script>` tags and uses the Slimit Javascript parsing library to parse and rewrite that code. Parsing first converts the Javascript code into an AST, and then rewriting is done by using a visitor pattern, as implemented in `zoobar/lab6visitor.py`. The code in `zoobar/lab6visitor.py` contains a separate method for each AST node, which is invoked recursively: a parent AST node calls `self.visit()` on child nodes. Each such method is responsible for returning a string object representing the rewritten Javascript code. The initial code in `zoobar/lab6visitor.py` that we provide does not modify the Javascript at all, and simply prints back the exact code corresponding to the AST.
- Third, to help with Javascript rewriting, the `zoobar/htmlfilter.py` code adds a trusted library (included inline at the top of `zoobar/htmlfilter.py` as `libcode`). This library exports trusted interfaces that the rewritten code can access, such as a safe way of accessing the profile's DOM objects.

We have constructed a number of test cases to help you debug your Javascript sandboxing system. They are stored in `profiles`, and include the sample profile above with the annoying scrolling message (`demo.html`), an automated test case checking that this example profile works (`good-all.html`), and thirteen different malicious profiles that you will need to confine (`bad-00-eval.html` through `bad-13-event.html`).

You can invoke the HTML / Javascript rewriter by running `zoobar/filter-test.py`; it reads profile code as input and prints out sandboxed HTML and Javascript. For example:

```
httpd@vm-6858:~/lab$ ./zoobar/filter-test.py < ./profiles/bad-00-eval.html
...
var s = "window.location = 'http://localhost:8900/test-bad';";
eval(s);</script></div>
httpd@vm-6858:~/lab$
```

To isolate Javascript, you will take the following approach:

- First, you will rewrite all function and variable names to prepend a unique prefix, `sandbox_`, to ensure that the code cannot directly access any functions or objects natively exported by the browser (such as `eval()`, the `window` object, the `document` object, etc). This way, if the code tries to invoke `eval()`, the rewritten version will invoke `sandbox_eval()`; since that function is not defined (or defined by the sandboxed code to point to some other sandboxed code), invoking that function will not allow escaping from the sandbox. Note that attributes like `bar` in `foo.bar` do not get prefixed. Neither do unquoted identifiers in object literals, like `{ x: 1, y: 2 }`.

  This will break all the tests until you also extend the trusted library (`libcode`) to support the `setTimeout()` Javascript function and the `textContent` property of DOM elements. The original functions are no longer accessible prefixed.

Be sure to guard against possible attacks through these interfaces; the native `setTimeout()` function allows specifying the callback object either as a function or as a string (which then gets `eval()`ed). We use `textContent` over `innerHTML` because `textContent` creates a text node directly without interpreting as HTML, so you needn't sanitize the input for additional `script` tags.

- Second, you will need to prevent the sandboxed code from accessing dangerous properties of objects. For example, each object in Javascript has a property that gives access to that object's *prototype*, which you can think of as being similar to the object's "class" in traditional object-oriented languages. Having access to the prototype allows changing the methods of that prototype ("class"). For example, an adversary can change the `substring` method of *all* strings by doing something like:

```
var s = "any string";
s.__proto__.substring = function() { return "gotcha!"; };
```

This is dangerous because it affects how other objects in the system behave, including objects used by other (trusted) Javascript code in the same page. Other methods allow indirect access to `eval`-like functionality, such as the `Function` constructor:

```
var f = function() { return 0; };
var newfunc = f.constructor("alert(document.cookie);");
newfunc();
```

And `__defineGetter__` may be called outside an object, in which case you define variables in global scope, which can confuse the outside code:

```
var f = [].__defineGetter__;
f("foo", function () { ... });
```

To prevent these attacks, you will need to find all instances where an object's property is accessed (as `objname.propname`), and check that `propname` is not one of the dangerous attributes: `__proto__`, `constructor`, `__defineGetter__`, and `__defineSetter__`. If it is, replace it with `__invalid__` or so. Raising an exception will also work, but the tests will be unhappy.

- Third, you will need to ensure that dangerous attributes cannot be accessed using array-like brackets (e.g., `object['__proto__']`), even if the array index inside of the brackets is a variable whose value is only known at runtime. To do this, we suggest using the same trick as FBJS uses: rewrite statements like `o[i]` to `o[bracket_check(i)]`, where `bracket_check()` is a Javascript function that you include in your trusted `libcode`, which checks if its argument is one of the dangerous attributes (listed above), and if not, returns its argument verbatim.

  Be careful, in the implementation of `bracket_check`, of objects with custom `toString` or `valueOf` methods. An adversary can pass in an object which stringifies as a safe string the first time, and as a blacklisted string the second.

- Finally, you will need to work around JavaScript's handling of the `this` keyword. If a function is called directly, as opposed to as a property of an object, `this` refers to the global object (in a browser, this is `window`). This would allow the sandboxed code to access unprefixed global variables. We recommend using a similar trick to FBJS: rewrite instances of `this` to a call to `this_check(this)`, where `this_check` returns `null` if its argument was `window`.

There are some aspects that this lab does not require you to get right. For example, each function has a `bind` method, which by default is `Function.prototype.bind`. This method in turn has an `apply` method, `Function.prototype.bind.apply`, which could be legitimately used by (un-sandboxed) Javascript code. However, with the current sandboxing scheme, a sandboxed piece of Javascript code could modify `Function.prototype.bind.apply` by creating some function `var f = function() {};` and then assigning to `f.bind.apply = ...`. The right solution for this problem is to call `Object.freeze()` on such shared objects and their prototypes, but we don't require you to do this for this lab.

Exercise. Implement Javascript sandboxing as described above. You will need to modify `zoobar/lab6visitor.py` and `libcode` in `zoobar/htmlfilter.py`.

Make sure that your sandbox works correctly with the `demo.html` profile, and stops the attacks in `bad-*.html` profiles. You can test this profile code by uploading it into (and viewing it through) the zoobar site on your VM. Alternatively, you can manually test it by running the profile code through `./zoobar/filter-test.py` (as shown above), and then loading the resulting HTML code in your browser. It will redirect to a URL containing either `test-ok`, `test-bad`, or `test-broken`.

You can check whether your system works correctly by running `make check`. This uses the PhantomJS JavaScript engine, which should produce the same results as actually running it in Firefox.

You are done! Run **`make submit`** to upload your answers.

6.858 Computer Systems Security

Fall 2014