Seminar: Cryptographic Protocols
Staged Information Flow for JavaScript

Sebastian Wild

1 Introduction

JavaScript is an important component of modern web applications since it is possible to execute code after the content of the homepage was sent to the browser. Furthermore libraries can be downloaded during run time from different sources across the web. After the Web 2.0 era was proclaimed, it's popularity rose much more. But there is also a downside to the flexibility that JavaScript provides. It has few protection or information hiding mechanisms, which leads to new security vulnerabilities such as cross-site scripting and code-injection attacks.

At first we want to examine the problem on a real example. One possible

```
<script src="http://adnetwork.com/insert-ad.js">
<textbox id="SearchBox">
<button id="Search" onclick="doSearch()">
<script type="JavaScript">
var doSearch = function() {
  var searchBox = document.nodes.SearchBox.value;
  var searchStr = searchUrl + searchBox;
  document.location.set(searchStr);
}
</script>
```

Figure 1: A snippet of JavaScript based on www.wsj.com. When the user clicks the Search button, the doSearch function appends the contents of the SearchBox to a base URL string searchUrl, and redirects the page to the resulting URL.

vulnerability was found in a study conducted by Google in [2]. As in our example in Figure 1, a lot of web pages include advertises into their content. Those advertises are usually dynamically loaded and distributed by ad agencies. But the ad does not necessarily come from the first ad agency. The first agency might include JavaScript code from a second level agency. This can go through several tiers of agencies. In their study, Google found a case were code was included from a reputable and non-malicious American ad agency. After several tiers of indirections, JavaScript code was included from a malicious ad provider in Russia. We want to examine a possible attack on our example in Figure 1. In the first line of our example a JavaScript is included which provides us with the advertisement. After that we have a text box and a button which provides us with a search mechanism in combination with the doSearch Function in the JavaScript at the end of the Figure. If the ad script would include JavaScript from a malicious provider, the malicious script could overwrite the value of the

global variable searchUrl, redirect the user to a malicious site and use a browser vulnerability to compromise the client's machine. In this scenario the attacker can direct the user to a malicious site without directly changing the document's location.
In this scenario we can observe one problem of third party code, it can affect sensitive data. In this case, a variable was reset and the user was redirected to a malicious site. But the attacker could also read information, e.g. from cookies, and use that information to attack the client. Therefore a information flow policy is suggested in [1], which ensures that sensitive data can not be written/read from third parties. But JavaScript has some features which make it difficult to analyze the code. Those are:

- dynamic typing: Variables do not have a type, the type depends on the value assigned to the variable
- first-class functions: Functions are objects which have properties, methods and as in functional programming languages there is basically no difference between a function and a variable
- objects: Properties and their values can be added, changed or deleted during run time
- prototype: JavaScript does not have classes and it uses prototypes

In addition to those features, since the code is not available until it is executed it is not possible to use a static analysis and therefore [1] proposes a staged approach. The first component, the JavaScript staging analysis runs on the server. We pass it the context and our policy as input. The policy defines two sets of variables. The confidentiality policy defines the set of variables which should not be read and the integrity policy defines the set of variables which should not be written to. With that information, the server computes the residual policy which contains the must-not-read and must-not-write variables. The actual test if a third-party JavaScript is safe to execute or not is conducted on the client or more precise the browser. The browser contains a component called residual policy checker. If the user requests a web page in his browser, the context is immediately sent to the JavaScript engine. The code is executed until it reaches the part of the code where the third-party code is included. The residual policy and the hole are handed to the residual policy checker which then has to check for each variable in the hole if it violates the must-not-read and must-not-write rule, specified in the residual policy. If the residual policy checker outputs that the script is save to execute then it is forwarded to the browsers JavaScript engine, otherwise the execution of the script is denied.

2 Information Flow Analysis

In this section we want to describe how the information flow is covered in the analysis. Before we get into details, we want to introduce two naming conventions.
The Context: The context is the JavaScript code which is provided by the owner of the web page.
The Hole: The hole contains the third party JavaScript code.

First we describe how the static analysis works using information flow and in the second part we describe the staged analysis. The practical use of both methods are shown in the section about the experimental results.

2.1 JavaScript Static Analysis

The analysis tracks the information flow of the code. This can be described by a graph where the nodes represent program constants, variables or functions. If we consider the expression $x = 0$ then this would be represented by two nodes, one for x, one for *0* and the assignment represented by an edge from the node *0* to the node x. A function is represented by a special node labeled *Fun* and two additional inner nodes, one for the arguments and one for the return values. For this kind of analysis we have to consider that even third-party code is available at run time, so that there is no code loaded dynamically. Out of the JavaScript code we can build the information flow graph. We have to look out for flows between the function definition and their call. The flow goes from the *Fun* node to a variable node which is labeled with the name of the function (since functions are variables in JavaScript, we do not need a new type of node) to another *Fun* node representing the function call. If a function has an argument, then the definition node contains the name of the argument variable and the call node contains the value which we pass to the function. There are also edges between the argument and the return value nodes of the function definition/call. If we have the complete graph including the hole we have to check if there is a flow from the third party code to the document location or from the document cookie to the hole. It is save to execute if there is no flow and otherwise it is not save to execute.

2.2 JavaScript Staged Analysis

In this section we talk about the proposed staged approach which has an advantage considering the execution time, but we come to that part later.

2.2.1 Computing Residual Policies

To compute the must-not-read-vars and must-not-write-vars we use the flow graph which we described in the previous section. In this case we taint the nodes and propagate a no write taint backwards since if a variable is not allowed to be written to another variable which is assigned to the variable must also not allowed to be written to. In the no read case it is then obvious that the taint has to be propagated forward. This approach is straight forward, but we have to look out for functions and aliasing.
Considering functions, we have two different cases. The first case regards the flow from a function node to a variable node where the argument node got tainted no write. The function node automatically gets tainted no read, since the function would write to a variable which is tainted as no-write variable. This could be the document.location and therefore we do not allow the hole to call the function. The second case regards an argument node which received a no read taint. For this case consider a function call *foo(document.cookie)*. If the hole would redefine the function foo it would get the document cookie as an argument and could read the information from the cookie. Therefore the

function node *Fun* has to be tainted as a no write node and this information has to be propagated backwards.
In the case of aliasing, the residual policy might miss future reassignment of variables. E.g. if we have the document cookie in our no read rule and the context could have the assignment $z = tmp.cookie$, the hole could now reassign tmp to document and read z. Therefore we have to make sure that if a field is tainted for some object, every field is tainted. Hence we have to add *tmp.cookie* and z to the no read rule.

2.2.2 Checking the Residual Policies

To check if a hole satisfies the residual policie, our residual policy checker checks if the variables of the hole are either in the rules for do-not-read-vars or do-not-write-vars. As you can see in the next section this can be done efficiently in practice.

3 Experimental Results

To verify that the theoretic approach can be used in practice, in [1] they describe a experimental setup and the results they get.

3.1 Experimental Setup

They implemented a Firefox extensions which collects dynamically loaded scripts from web pages and then in-lines the code in the surrounding context as it would have been there and we get a static context. As a front-end, JSure is used as a parser to generate an OCaml abstract syntax tree out of the JavaScript code. Out of the ABS constraints are generated which then are solved by the constraint solver Banshee. The Firefox extension is written in approximately 500 lines of JavaScript, 6000 lines of OCaml to generate the constraints and in addition to Banshee 400 lines of C code to solve the constraints. The web pages from the Alexa top 100 list were used to execute the benchmarks. 97 of those 100 sites used JavaScript and of those 97 sites 63 had dynamically loaded third-party code. Information flow was done on the cookie confidentiality, meaning that the cookie information could not be read by the third party code and secondly the integrity of the document location as explained in the example at the beginning.

3.2 Performance of the Analysis

The full analysis has proven to be scalable. In average the analysis took 9.9 seconds and approximately 80 % of the sites were analyzed in less than 12 seconds. There were just a few web pages where the analysis took longer and after taking a closer look, they found out that these sites used prototyping extensively. The Wall Street Journal page was the largest page which was analyzed with 43,698 lines of code and the analysis took 76.0 seconds. Considering the results we see that the full analysis is not applicable for real browser implementation. On the other hand staged analysis has the advantage that the residual policy can be calculated on the server as soon as the page is deployed by the developer. Experiments showed that this part takes 14 seconds in average. The client side of the analysis then just checks the must-not-read-vars and must-not-write-vars which can be done very fast, it takes just 0.13 seconds in average. So the staged approach is a way to analyze JavaScript in browsers which would not affect the time it takes to load a web page in a serious manner.

4 Summary

The experimental results show that even the full analysis of JavaScript scales for large scripts but it is not suitable in practice. On the other hand the staged analysis with the residual policy checker is fast enough on the client side such that it could be implemented. That would require not only an implementation on the browser side but also an additional module for web servers. Modern browsers use different techniques to ensure that the execution of JavaScript code is save. For example Google implemented a virtual machine for the execution of JavaScript code such that the code can not affect the clients computer outside of the sandbox. For future work it would be interesting to compare those approaches for security and performance.

References

[1] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[2] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.