POSTER: Finding JavaScript Name Conflicts on the Web

Mingxue Zhang
Chinese University of Hong Kong
mxzhang@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

Yi Wang
Southern University of Science and Technology
wy@ieee.org

ABSTRACT

Including JavaScript code from many different hosts is a popular practice in developing web applications. For example, to include a social plugin like the Facebook Like button, a web developer needs to only include a script from facebook.net in her/his web page. However, in a web browser, all the identifiers (*i.e.*, variable names and function names) in scripts loaded in the same frame share a single global namespace. Therefore, a script can overwrite any of the global variables and/or global functions defined in another script, causing unexpected behavior.

In this work, we develop a browser-based dynamic analysis framework, that monitors and records any writes to JavaScript global variables and global functions. Our tool is able to cover all the code executed in the run time. We detected 778 conflicts across the Alexa top 1K websites. Our results show that global name conflicts can indeed expose web applications to security risks.

CCS CONCEPTS

• **Security and privacy → Browser security**; **Web application security**.

KEYWORDS

JavaScript; Name conflicts; Web applications

ACM Reference Format:
Mingxue Zhang, Wei Meng, and Yi Wang. 2019. POSTER: Finding JavaScript Name Conflicts on the Web. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363268

1 INTRODUCTION

It is very common to separate code of different functionalities into multiple JavaScript files in today's web applications. Including JavaScript code from other hosts is also a very popular practice in developing web applications, because a developer can reuse the code in other third-party programming libraries and easily build an application rich of functions.

While enhancing the functionality of a web application, the included third-party scripts may cause unexpected behavior to the developer's own code. In the client-side JavaScript runtime environment, *i.e.*, the web browser, there exists a single global namespace for all identifiers (*i.e.*, variable names and function names) in scripts loaded in the same frame. Any variable or function defined in a script's own main scope is available to any other script running in the same frame. This means that a script can not only directly call global functions and read the values of global variables in another script, but also overwrite any of the global variables and/or global functions. Since JavaScript is a weakly typed programming language, a script can even change the type of any global variable without causing any exceptions or errors. Such kind of global name conflicts can compromise the integrity of the developer's own code and the third-party library code. As a result, a script may take a different branch, return an incorrect value, or simply crash, *etc.*

Even if a developer carefully examines the source code before she/he includes a third-party script, which could be very difficult because of code minimization or obfuscation, global name conflicts cannot be avoided. The third-party code is hosted on a remote server and can be modified by the script provider at any time without notifications. Further, a script can dynamically include any other scripts, which may also contain global names that conflict with the existing ones. They might be prohibited by the Content-Security Policy (CSP) [9]. However, it had been shown that CSP had a very low adoption rate [10] because many websites need to load additional scripts from almost any sources.

Prior research have studied potential global name conflicts between two JavaScript libraries. In [5], the authors tested if two libraries would cause different behaviors when they were loaded in different settings. They created a synthetic client for each of the settings. Such clients can test the simple operations of the libraries. They may not well represent the code in real applications that could be much more complex. Further, only a limited number of libraries were studied in a synthetic environment. The result cannot reflect the conflicts in real applications, which may include more than two libraries. Finally, the analysis was based on a selective record-replay dynamic analysis framework [8] that instruments specified source code. Thus, the tool does not cover any code that is loaded dynamically.

In this work, we develop a browser-based dynamic analysis framework that can monitor and log writes to JavaScript global memory locations (*i.e.*, variables and functions). In particular, we instrument JavaScript code dynamically by modifying the V8 JavaScript engine of the Chromium browser. We insert our monitoring code to log any operations that are related to memory write in a script. This allows us to cover all code executed at the run time. With the logs, we are able to detect three kinds of global name conflicts – 1) variable value conflict, where two or more scripts write different values of the same type to the same global variable; 2) function definition conflict, where multiple scripts define a global function with the same name; and 3) variable type conflict, where multiple scripts assign values of different types to the same global variable.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363268

We implemented a prototype of our framework based on the version 71 of Chromium. We conducted a measurement study by using the prototype to collect logs from the main pages of the Alexa top 1,000 websites. In total, we detected 47 variable value conflicts on 25 websites, 728 function definition conflicts on 85 websites, and 3 variable type conflicts on 2 websites.

2 DESIGN AND METHODOLOGY

We illustrate how our analysis framework works in this section. We record each function definition in the V8 parser to detect function definition conflicts (§2.1). We dynamically instrument any JavaScript code that is executed to log writes to a global variable (§2.2). Our instrumentation dynamically infers the type of the write target for each write operation. The logs allow us to detect conflicting writes by different scripts to the same global functions or the same global variables (§2.3).

2.1 Recording Global Function Definitions

The root cause of function definition conflicts is two or more scripts define their functions using the same global function name. Therefore, we need to find all functions that are explicitly defined in each script. Whenever a global function is parsed by the V8 parser, we log its name and the URL of its script. Especially, we log only the ones with a non-empty function name. This allows us to detect multiple explicit definitions of the same global function.

2.2 Recording Writes to Global Variables

We focus on four types of operations that JavaScript may perform to write to a variable: 1) assignment statements; 2) object literal expressions; 3) call expressions; and 4) return statements. When any of such operations is executed, our framework uses the `typeof` operator in JavaScript to infer the type of the write target. It also records the value of the target if it is a primitive type variable, a unique ID for each operation *e.g.*, the ID of the script, and the ID of the execution context (frame). Except for the above data, we collect additional information for each kind of operations as we describe next.

Assignment Statements. For each write target `v` in an assignment statement, in order to tell if `v` is a global variable, our tool checks all the declared variable names (including parameters, if the current scope is a function scope) within the scope of the current assignment statement. It continues searching in the outer scopes until a match is found or it reaches the global scope. If no match is found for a variable `v` or it is found only in the global scope, it is considered as a global variable.

Object Literal Expressions. A script may change a property of a global object instead of overwriting the entire variable. Therefore, our tool needs to record writes to object properties. First, it logs direct assignments to object properties such as $o.p = 1$. It also records a special kind of writes to object properties – object literal expressions. Specifically, for each object literal expression, *e.g.*, $\{p_1 : e_1, \ldots\}$, it logs a write to $o.p_1$ where o is the temporary variable representing the object literal. We do realize that our approach is not comprehensive, which we will discuss in §4.

Call Expressions. An object can be passed by reference as an argument to a parameter of a function and then be modified within the function through the parameter variable. Therefore, we need to keep track of the pass of objects in function calls. Specifically, for each function call, our tool logs the function name and a list of arguments that are passed into this function. When the program enters into the function body (*i.e.*, the callee), it records each parameter.

In this way, we can find a corresponding parameter record logged in the callee for each of the argument record logged in the caller. If a write to the local parameter variable is recorded, we can then trace back to the caller logs to determine if this would cause a global name conflict.

Return Statements. A local object u may be initialized within a function and then returned and assigned to another variable v. This variable may be modified later by the caller function or by any other functions if it is a global variable. To detect potential conflicts, we also record which local variable is being returned in a return statement. This allows us to link the writes to u with the writes to v and to detect conflicting writes to the same (global) object.

2.3 Detecting Conflicts

In this section, we explain how we detect conflicts using the logs described in §2.1 and §2.2.

Alias Analysis. For each write to a variable, we will maintain an alias if the write is a copy-by-reference or a pass-by-reference operation. An alias is removed when one of the variables is assigned with another object. Then, we will find the write records of the current variable as well as the records of its aliases to determine if there exists a conflict.

Function Definition Conflicts. To find function definition conflicts, we check the function definition logs in each frame to find if the same global function had been defined for more than once by different scripts.

Value Conflicts and Type Conflicts. If a global variable is of a primitive type, it does not have an alias. We will search any other write records to the same global variable. If the logged values in two records are different and the writes are performed by two different scripts, we report it as a variable value conflict. However, if the types of the global variable are different, we report it as a variable type conflict.

If a global variable is an object, a value conflict may happen when the variable itself is overwritten with another variable, or a property of the object is written. Therefore, except for the assignment records to the same variable, we also search the write records of all the object's valid aliases with regards to the current assignment. For writes to the object variable itself, a value conflict is reported if it is assigned with another object, and a type conflict is reported if it is assigned with a primitive-type value. For writes to the property of the object, we apply the above rules depending on its type.

3 EVALUATION

We crawled data from the main pages of the Alexa top 1K websites in August, 2019. We gathered 957 function definition log files and variable write log files from 957 frames loaded on 893 websites. We were not able to collect data using our current implementation

```
function createCookie(a, e, b) {
  if (b) { var d = new Date;  ..... }  else b = "";
  document.cookie = a + "\x3d" + e + b + "; path\x3d/"
}
```

```
function createCookie(b, c, a) {
  if (a) { var d = new Date;  ..... }  else a = "";
  - 1 < google_tag_manager["GTM-KBNVHH"].macro(134).indexOf("
      zoho.eu") ?
  document.cookie = b + "\x3d" + c + a + "; domain\x3d.zoho.eu;
      path\x3d/" : ..... ;
}
```

Listing 1: Conflicting definitions of `createCookie()` in different scripts on https://www.zoho.com.

from the rest websites. We leave it as a future work to improve our implementation. Except for those that were extremely large (with over 1 million records), we were able to analyze 947 (98.96%) assignment log files and 957 (100.00%) function definition files.

In summary, we found 47 variable value conflicts on 25 websites, 728 function definition conflicts on 85 websites, and 3 variable type conflicts on 2 websites. Note that if a conflict was caused by the same script, we do not report here.

Interestingly, we found 46 cookie-related functions were overwritten by at least one script. One example was detected on website https://www.zoho.com, where 5 inline scripts all defined a global function `createCookie()`. The definitions from different scripts are slightly different, as shown in Listing 1.

Similarly, we discovered multiple definitions of function `getCookie()` on https://zoom.us/. This shows that a JavaScript global name conflict could expose a victim user to security risks. For example, a malicious third-party can manipulate `getCookie()` to force a user to use the attacker's session in the client-side code and trick the application code into processing the attacker's Cookie.

4 DISCUSSION AND FUTURE WORK

We now discuss the limitations of our current work and our future work.

Incomplete Support of Objects. An alternative way to define an object property is to initialize it through the identifier `this` within the constructor or a method of an object. For example, $t = new\ Obj(...)\ \{this.p = e;\}$. In order to determine the object that `this` refers to, we need to know the receiver object of the methods. We plan to support it in our future work.

Function Definition Conflicts. Except for directly declaring a global function, a script can also assign a function literal to a global identifier, *e.g.*, $f = function()\{...\}$. This could result in a function definition conflict or a type conflict. To detect this kind of conflicts, we need to also cross check the function definition logs and variable write logs. We will include this analysis in our future work.

Characterization of Conflicts. Our categorization of the detected conflicts is not sufficient for comprehensively investigating the problem of JavaScript global name conflicts. For example, it would be interesting to analyze the conflicts of third-party scripts overwriting first-party defined names. We aim to perform a comprehensive analysis of the conflicts in the future.

5 RELATED WORK

JavaScript Conflict Analysis. Patra *et al.* proposed ConflictJS, an automated approach to analyzing the conflicts between JavaScript libraries using synthetic clients [5]. They considered simple operations like direct variable write and property write, and studied limited number of JavaScript libraries. In [8], the authors proposed a dynamic JavaScript analysis framework that is based on selective record-replay technique. Therefore, the tool is not able to cover dynamically loaded code. In contrast, our dynamic analysis framework is able to detect the conflicts between scripts that are even dynamically loaded.

JavaScript Type Inference. Pradel *et al.* proposed TypeDevil to detect type inconsistency in JavaScript [6]. Jensen *et al.* defined a type analysis for JavaScript based on abstract interpretation [3]. Hackett *et al.* presented a hybrid type inference approach for JavaScript based on points-to analysis in [1]. These works focus on inferring JavaScript type information within a single script. Meanwhile, there have been several learning-based approaches to predicting the type for JavaScript code [2, 4, 7]. They aimed to statically infer about a variable type and therefore enable the generation of much faster code, which is orthogonal to our work. In our work, we leverage the JavaScript built-in type checker to infer the type of a variable at run time.

6 CONCLUSION

We developed a browser-based dynamic analysis framework to study JavaScript global name conflict problem on the Web. We collected data from the Alexa top 1K websites. In total, we detected 47 variable value conflicts on 25 websites, 728 function definition conflicts on 85 websites, and 3 variable type conflicts on 2 websites. We further investigated the detected conflicts and demonstrated that the global identifier conflicts may lead to security issues.

ACKNOWLEDGMENT

The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 24209418).

REFERENCES

[1] Brian Hackett and Shu-yu Guo. 2012. Fast and precise hybrid type inference for JavaScript. *ACM SIGPLAN Notices* 47, 6 (2012), 239–250.
[2] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.
[3] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.
[4] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada.
[5] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden.
[6] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.
[7] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.
[8] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Saint Petersburg, Russia.
[9] W3C. [n.d.]. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.
[10] Ming Ying and Shu Qin Li. 2016. CSP adoption: current status and future prospects. *Security and Communication Networks* 9, 17 (2016), 4557–4573.