# Virtual Browser: a Web-Level Sandbox to Secure Third-party JavaScript without Sacrificing Functionality

Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, and Yan Chen
Dept. of EECS, Northwestern University
Evanston, IL, USA
yinzhicao2013@u.northwestern.edu, lizc@cs.northwestern.edu,
vrastogi@u.northwestern.edu, ychen@northwestern.edu

## ABSTRACT

Third-party JavaScript offers much more diversity to Web and its applications but also introduces new threats. Those scripts cannot be completely trusted and executed with the privileges given to host web sites. Due to incomplete virtualization and lack of tracking all the data flows, all the existing works in this area can secure only a subset of third-party JavaScript. At the same time, because of the existence of not so well documented browser quirks, attacks may be encoded in non standard HTML/JavaScript so that they can bypass existing approaches as these approaches will parse third-party JavaScript twice, at both server and clint side.

In this paper, we propose Virtual Browser, a completely virtualized environment within existing browsers for executing untrusted third-party code. We secure complete JavaScript, including all the hard-to-secure functions of JavaScript programs, such as *with* and *eval*. Since this approach parses scripts only once, there is no possibility of attacks being executed through browser quirks. We first completely isolate Virtual Browser from the native browser components and then introduce communication by adding data flows carefully examined for security.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection—*Information flow controls, Invasive software*; D.1.0 [**Programming Techniques**]: General

## General Terms

Security, Design, Language

## Keywords

Third-party JavaScript, Web Security, Virtualization

## 1. INTRODUCTION

Modern websites tend to use third-party JavaScript to enrich their functionalities. Web mashups, which combine services from existing websites, has become more and more popular. For example, a website may use Microsoft's Bing Maps API for providing location information, employ Google Analytics to track its visitors' behavior, include JavaScript code from targeted advisement companies for increasing revenue, and enable third-party widgets for richer functionalities. In each of these cases, some third-party JavaScripts, which are not developed by the website, has privileges to execute even from the same origin as the JavaScript from the website itself. Although these third-party JavaScripts enrich the functionalities of the website, malicious third-party Javascripts can potentially fully subvert the security policy of the website, and do all kinds of attacks.


*CCS'10,* October 4–8, 2010, Chicago, Illinois, USA.
ACM 978-1-4503-0244-9/10/10.

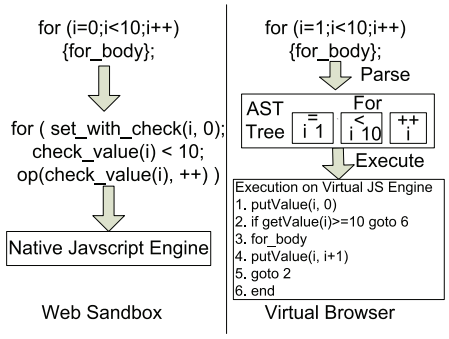


**Figure 2: Comparison of Web Sandbox and Virtual Browser when Executing For-loop**

Our virtualization technique is significantly different from the existing proposed ones, for example, Web Sandbox [7] and Google Caja [5]. The key difference is whether third-party JavaScripts are directly running on a native JavaScript engine. Figure 1 clearly shows the differences. We execute third-party JavaScripts on our virtualized JavaScript engine; on the other hand, existing approaches check the parameters of each third-party JavaScript expression and then let them execute directly on the native JavaScript engine. Web Sandbox [7] makes a big step on virtualization. It provides a virtualized environment for native execution of third-party JavaScript, but its execution is still restricted by the parameter checking model. As shown in the left side of Figure 2, they provide a virtualized environment for *for* loop, but the *for* loop itself is still running on a native JavaScript engine. This is also the reason why they cannot deal with complex JavaScript functions such as *eval* and *with*. For *eval*, their parameters can be unparsed JavaScript statements. For *with*, without executing the scripts themselves, it is hard to know the execution context.

A Virtual Browser is a virtualized browser built on top of a native browser. The idea of virtual browsers is comparable to the idea of virtual machines. A virtual browser is written in a language that a native browser supports, such as JavaScript. It has its own HTML parser, CSS parser and Javascript interpreter, independent from the native browser. Third-party Javascript is only parsed once in the virtual browser and run on the virtual JavaScript interpreter. The third-party JavaScripts are isolated from the JavaScripts of the website by design. Then, we introduce the necessary communications between JavaScripts from the website and third-party JavaScripts.

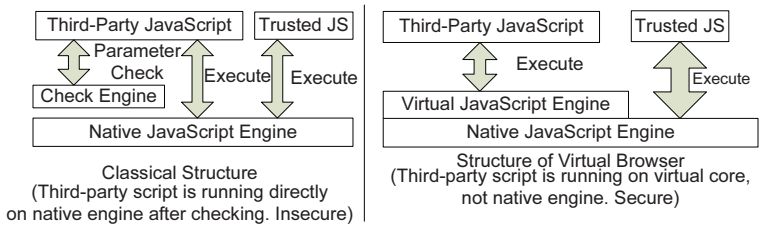


**Figure 1: Classical Structure (left) v.s. Simplified Structure of Virtual Browser (right)**

```
1. <div id = "my_div1" style="…">
2.    <scr
3. ipt>
4.       var JSON_str = "…";
5.       eval(JSON_str);
6.       setTimeout("alert('hello');", 10);
7.         ….
8.    </script>
9.    <div id = "my_div2>
10.   <p> abcd
11.   <p> cdef</p>
12.   …
13.   </div>
14.</div>
```

**Figure 3: A Motivating Example**

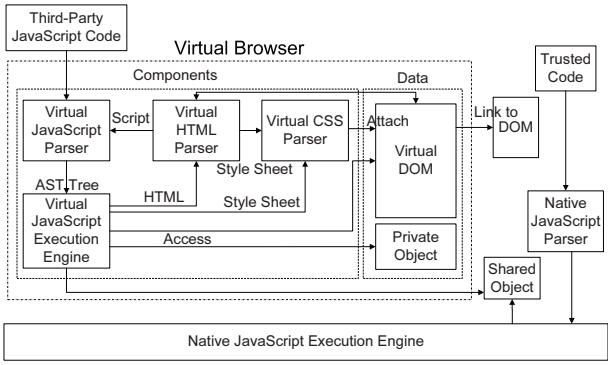


**Figure 5: System Architecture**

## 2. A MOTIVATING EXAMPLE

We begin with a motivating example in Figure 3. We discuss two important points using this example: (*a*) some complex functions cannot be implemented in existing sandboxing approaches, and (*b*) some erroneous programs are not recognized by existing approaches. Lines 4-5 show a JSON string being executed by *eval*, a complex function not protected by present approaches. Developers however still use this method to parse JSON strings in old browsers which do not have native JSON support. Line 6 shows us another example about *setTimeout*. As shown in Figure 1, the classical runtime approaches (such as runtime part in GateKeeper [6]) employ a parameter checking model, implying that they cannot check the safety of some complex but useful functions, such as *eval* and *setTimeout*, whose parameters need to be passed to the JavaScript parser. Without incorporating execution tracing, it is very hard to determine by plain parameter checking if these parameters are malicious. Web Sandbox [7] also adopts parameter checking as mentioned above but additionally creates a virtualized environment for the third party scripts in which the variables have a different namespace than what is visible to the native engine. However, the arguments of *eval* when generated dynamically would bypass this instrumentation. Since the execution is still done on the native JavaScript engine, *eval* cannot be safely executed in Web Sandbox's approach. Static approaches have even greater problems; they cannot even ensure if the parameters are checked at all. For example, an *eval* call obfuscated as "*a=eval;a.call("alert(1)");*" is not easy to detect using static approaches. There are many similar obfuscating techniques discussed by Guarnieri et al. [6]. However, 44.4% of the websites use *eval* [10] and about 9.4% of widgets use *with* [6]. We cannot simply ignore the existence of these complex functions.

Lines 2-3 and Line 10 of Figure 3 give some examples of bugs in the third-party programs. Because web-programmers are humans, mistakes are very likely during programming. This is one reason why existing web browsers all support non-standard HTML and JavaScript, commonly referred to as browser quirks. Browser quirks are well studied in BluePrint [9]. Because of the prevalence of browser quirks, server side checkers may interpret third party content differently from the clients. Considering the quirk in lines 2-3 of Figure 3 as an example, even if the server side checker may pass it as an unknown tag, if the client browser consider it as a script tag, it will cause unsupervised script running at client side. Vulnerabilities and attacks of this kind are very common. For example, the server side filter on the facebook server had such a vulnerability [8]. A string *<img src="..." onload:=attackcode>* is interpreted as *<img src="..." onload = attackcode>* at browsers (Firefox 2.0.0.2 or lower) but as *<img src="...">* at server side filter. Also, in the notorious MySpace Samy worm [2], the server side script checker failed to recognize JavaScript embedded in CSS codes but browsers executed the embedded JavaScript because of the difference in the order of the invocation of the JavaScript parser and the CSS parser. All existing approaches, either static, such as GateKeeper [6] (mostly static) and ADSafe [1], or runtime, such as Web Sandbox [7] and Google Caja [5], define a particular server side interpretation which may be very different from the browsers' interpretation, and hence remain vulnerable to such attacks.

## 3. DESIGN

### 3.1 Architecture

The architecture of Virtual Browser is shown in Figure 5. Virtual Browser is very similar to a native web browser except that it is written in JavaScript. Virtual Browser is composed of two parts, components of Virtual Browser and data in Virtual Browser. Components of Virtual Browser is consisted of virtual JavaScript parser, virtual JavaScript execution engine, virtual HTML parser, virtual CSS parser, etc. Data in Virtual Browser is consisted of virtual DOM, private object, etc.

The same as native web browser, virtual JavaScript/HTML/CSS parser is used to parse JavaScript/HTML/CSS code. Virutal JavaScript execution engine is used to execute parsed JavaScript AST tree. Virtual JavaScript execution engine and HTML parser use virtual DOM methods in virtual DOM to access its contents. Private data is used to store JavaScript object in virtual browser.

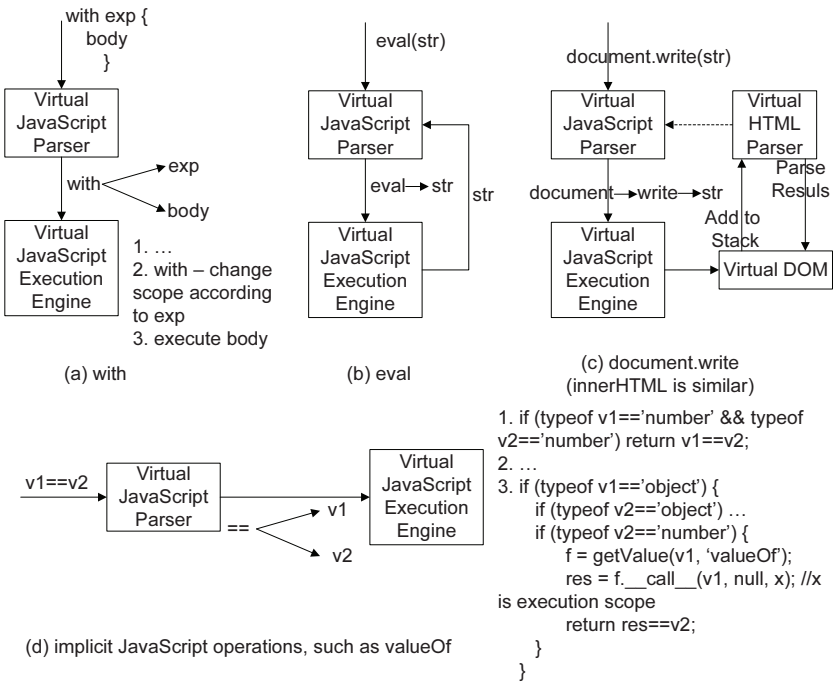


**Figure 4: Examples of Several Cases**

## 3.2 Design of Interfaces of Components

The most important interface is the interface of JavaScript Core, which has three parts, putValue, getValue and function call/return. putValue is loaded every time an object is changed. Every modification to a private (from third-party) function/variable or a native (from trusted scripts) function/variables goes through putValue. getValue provides a interface for every read operation. Function calls are used for calling shared functions from natively running code and private functions from third-party codes. Our design of the interface of the virtual JavaScript engine is similar to the one of the native JavaScript engine. Several works [3, 4] have details about the native JavaScript engine's interface.

## 3.3 Design of Communication between Components

We will present how third-party JavaScript codes flow inside Virtual Browser. When third-party JavaScript codes come into Virtual Browser, virtual JavaScript parser will first parse it into AST tree and give the tree to virtual JavaScript execution engine. Virtual JavaScript execution engine will execute the AST tree as normal JavaScript interpreter does. When a HTML content is found, virtual JavaScript execution engine will send it to virtual HTML parser. Similarly, JavaScript codes and CSS style sheets will be sent to virtual JavaScript and CSS parser. Virual HTML parser will parse HTML and will send scripts/style sheets to virtual JavaScript/CSS parser. All of these processes are shown in Figure 5.

## 3.4 Case Studies

In this section, we illustrate several examples that previous approaches cannot deal with.

**with** *with* is a notoriously hard problem in this area. All the existing work cannot solve this problem. In our system, *with* becomes quite simple because we interpret JavaScript ourself. For example, as shown in Figure 4(a), *with document* in our system is just a switch of current context.

**eval** *eval* is usually banned by existing approaches because it will introduce additional unpredictable JavaScript. As shown in Figure 4(b), In our system, we just need to redirect contents inside *eval* back to our virtual JavaScript parser. No matter how many *eval*s are embedded(such as *eval(eval(...(alert('1'))...)*), JavaScript is still executing inside our virtual JavaScirpt Core.

**document.write/innerHTML** *document.write* and *innerHTML* are related to HTML parser. As shown in Figure 4(c), when our JavaScript execution engine encounters functions/variables like these, we will redirect them to HTML parser by calling methods in virtual DOM. All the traffic is still in control of our system.

**implicit JavaScript operations** Some of JavaScript operations are implicitly called. For example, when executing $v1 == v2$, *valueOf* may be implicitly called, as shown in Figure 4(d).

**arguments** *arguments* is implemented inside function. Arguments of current function is stored in current running context. When we use *arguments*, we can fetch it directly.

# 4. REFERENCES


[1] AD safe. http://www.adsafe.org/.

[2] Myspace samy worm. http://namb.la/popular/tech.html.

[3] Webkit source codes. http://webkit.org/building/checkout.html.

[4] BARTH, A., WEINBERGER, J., AND SONG, D. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *the 18th USENIX Security Symposium* (2009).

[5] GOOGLE. Google caja. http://code.google.com/p/google-caja/.

[6] GUARNIERI, S., AND LIVSHITS, B. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *18th USENIX Security Symposium* (August 2009).

[7] LABS, M. L. Websandbox. http://websandbox.livelabs.com/.

[8] SOTIROV, A. Blackbox reversing of xss filters. *RECON* (2008).

[9] TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (May 2009).

[10] YUE, C., AND WANG, H. Characterizing insecure javascript practices on the web. In *WWW '09: Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), ACM, pp. 961–970.