Sandboxing Third Party Code in a Website

1. Introduction

JavaScript is a scripting language that allows website developers to implement complex features in their websites. In fact, any time you visit a web page that does anything more than display static content, it is a safe to assume that JavaScript is involved. To name a few of its capabilities, it can read and write changes to a page's contents, make network requests, perform page redirections, and even manipulate standard built-in objects shared throughout the website. These capabilities, useful for website developers, are exploitable in the hands of a potential attacker. Unfortunately, certain aspects of modern web development make it too easy for them to gain access to it.

As early as 2009, at least 66.4 percent of popular websites were already including third party [1] and in 2018, that number had risen to around 86.8 percent [6]. In many (and from personal experience as a web developer I would venture to guess the vast majority) of these cases, developers are making a blind leap of faith. They are not only trusting that the code's original writers have no malicious intent, but also that the code they are receiving has not been altered or even completely changed by an attacker. The historically famous Sony rootkit attack [7] shows that faith in the former, and the recent "event-stream" (a popular third party module with 190 million installs) supply chain attack [8] shows that faith in the latter, are both misplaced.

There are many reasons a developer may choose to include third party code on their websites but there is only one reason for the blind trust they give it: because they do not have much choice. Current browsers simply do not provide web developers a way to restrict the capabilities of third party code included into their sites and by extension, the damage it can cause.

Waiting for browsers to change is not an option. Developers have needed a way to restrict code running on their sites for years and so far, browser vendors have done next to nothing to solve the problem. Asking developers to stop using third party code is not an option either. For many developers its benefits outweigh the dangers and in some situations, they have no other choice. Thus, the goal of this project is to search for, find, and test ways in which developers can create websites that are as secure as possible despite the inclusion of third party code **today**.

2. Background

A browser is an application that runs on a user's computer for the purpose of viewing resources on the web. A Uniform Resource Locator (URL) is an address pointing to a unique resource. Given a URL, the browser fetches the resource from an internet connected server and displays it to the user. A website is one type of web resource, consisting of instructions written in languages such as Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript, and data such as images and videos. The browser takes these instruction and data and uses them to display the website. The contents of a website are not limited to the resource served from its URL and during its runtime, it can dynamically include resources from other URLs.

Browsers provide three categories of resources to JavaScript executing on a website. The first category is access to Web APIs. It is through these APIs that JavaScript can interact with the browser, giving it the ability to read and update the contents of a website, make network requests, play audio tracks, redirect the browser, write to the clipboard, etc. The second category is access to standard built-in objects. These include fundamental objects (e.g. Boolean), indexed collections (e.g. Array), keyed collections (e.g. Map), dates, mathematical calculations, etc. The final category is access to the JavaScript engine itself. In a website, all JavaScript code is executed in a single thread and shares a stack and heap.

The most critical browser security mechanism is the Same Origin Policy (SOP). The SOP is an isolation mechanism that restricts how JavaScript in one origin can interact with resources in another. A resource's origin comes from the URL it was accessed at, and two URLs have the same origin if their protocols, hosts, and ports are the same. Given the URL `http://example.com/`, the protocol is `http`, the host is `example.com`, and the port is `80` (the default port for `http`). The SOP gives each origin access to its own Web APIs, standard built-in objects, and in some browsers, JavaScript thread. Communication between websites from different origins running in the browser is possible, but only through safe channels.

There is no browser security mechanism for restricting how JavaScript included onto a website can interact with the websites resources. From a security standpoint, the browser does not recognize any differences between the different scripts running on the same website. All scripts executing in its global environment share the same Web APIs, standard built-in objects, and JavaScript thread.

3. Requirements of a Secure Website

A secure website must have the following properties: confidentiality, integrity, authorization, and availability. A website has confidentiality if its non-public information is only accessible by authorized parties, integrity if its data, or software can only be altered by authorized parties, authorization if its computing resources can only be accessed by authorized parties, and availability if its information, services, and computing resources remain accessible for authorized use.

Without an access control mechanism, a website cannot have any of the properties listed above. The model for such a mechanism is described by the *reference monitor* concept. It consists of subjects: all parties that may access a resources, objects: all accessible resources, a policy: the set of rules defining the actions that subjects can perform on objects, and a reference monitor: a tamper proof, un-circumventable, and verifiable mechanism that validates a subject's request to perform an action on an object using the policy.

In order to minimize the violations of a website's security properties, following the security principle of least privilege, included third party code must be authorized to access only the fewest security sensitive resources needed for its task. That way, if a script does become compromised, the damage it can inflict is limited by the capabilities needed by the legitimate script. To enforce this authorization, an access control mechanism is needed to validate all requests made between third party code and a website's security sensitive browser resources.

4. Project Specific Requirements

Not all of the resources provided to JavaScript by the browser are security sensitive but it must be possible to restrict access to the ones that are. Access to any of the first category of a website's browser resources, the Web APIs, is dangerous and should be completely mediated. The dangers of access to the second category, the standard built-in objects, is more nuanced. Many of these objects are modifiable and an attacker can change their behaviour to affect code using them throughout the website. In addition, some of them can be used to perform Side-Channel-Attacks. Therefore, the ability to mutate any, and access some of these resources must be mediated. The danger of access to the final category, the JavaScript engine, is that it allows code to affect the availability of the rest of website in a Denial of Service (DoS) attack. However, given that even the SOP does not always protect different origin websites from these attacks, mediating access to this category is not required.

To be usable for developers today or in the near future, it must be implemented using existing and legacy browser technologies. Updates to browsers have historically been a slow process and as of today, there are no plans to provide a mechanism that meets our requirements. When updates finally do happen, they are generally opt-in changes, meaning that a website can only make use of them if the user viewing it has an updated browser. As a result, it will likely be years before developers will have and can rely on a browser-built solution.

The security mechanism must not depend on pre-processing third party code. If a security mechanism pre-processes code to remove restricted commands, it functions as a black-list since it defines what is not allowed. If a security mechanism fully isolates code and then gives it access to restricted commands, it functions as a white-list since it defines what is allowed. When designing security mechanisms, an important principle to follow is to favour using whitelist over blacklist mechanisms.

5. Related Work

The problem of trying to isolate code in a website is not a new one and many researchers and developers have taken a stab at solving it. The solutions that are relevant to the scope of this project can be divided into categories based on the mechanism they rely on. These categories are solutions relying on the Same Origin Policy, subsets of the JavaScript language, using a separate JavaScript engine, and using Web Workers. The following section will provide an example from each category.

5.1 Same Origin Policy

Solutions in this category protect a website by executing third party code under different origin than the website. The SOP mechanism handles the task isolating the third party code while still allowing it to communicate with the website through the `PostMessage()` function.

AdJail [2] seeks to solve the problem using different origin iframes. An iframe is an HTML element that allows an HTML document to load another inside it. If the iframe's document has a different origin than the parent document, the browser treats the code running on it as being from a different origin. AdJail runs third party code in an invisible, different origin iframe known as the shadow page. The iframe provides code with the exact environment it expects without giving it access to the main website's environment.

AdJail uses two scripts, script A running on the shadow page and script B running on the real page. All communication between the two pages is handled using the browser's `PostMessage` API. As the website executes, script A constantly monitors changes to the shadow page and communicates them to script B. Script B is then able to validate whether changes on the shadow page should be applied to the real page based on a security policy.

5.2 JavaScript Subsets

The Web APIs and standard built-in objects are all shared to code running on a site through global variables. In order to sandbox third party code executed under the same origin as the rest of the website, it must be prevented from directly accessing these variables. One way this can be achieved is by using an object-capability environment, in which executing code cannot access an object without an explicit and unforgeable reference to it. The full JavaScript standard does not qualify as an object-capability language but certain subsets of it do. Solutions in this category have the common feature of restricting untrustworthy code to one of these secure subsets of JavaScript.

JSand [3] uses a subset of JavaScript called Secure ECMAScript (SES) to realize its underlying object-capability environment. SES was discovered by researchers at Google who also created a JavaScript library that enables the execution of SES code in regular browsers. Initially, code executed by this library is not given a reference to any of the resources provided by the browser. References to necessary resources can be given to code executed by the library but care must be taken to avoid giving it unmediated access to browser resources.

JSand uses the JavaScript `Proxy` API to wrap browser resources before passing them to isolated code. These wrappers interpose between a caller and the resource they wrap and are capable of validating requests against a security policy. JSand also implements a membrane that transitively wraps objects returned by, and unwraps objects sent to a wrapped object.

5.2 JavaScript interpreters

This category of solutions involves executing third party code using a different JavaScript interpreter than the one used by the rest of the site. Code executed by a different interpreter is given its own set of standard built-in objects and none of the resources provided by the Web APIs.

Js.js [4] is a library that allows the execution of code in a JavaScript interpreter that runs on top of JavaScript. The prototype Js.js implementation compiles the Spider-Monkey JavaScript engine to LLVM, and from there into JavaScript to allow it to run in the browser.

Initially, the environment is given no access to the Web APIs but the Js.js library provides an API to allow developers to bind resources to the isolated environment. This makes it possible to pass objects to the environment that wrap browser resources and validate requests to them against a security policy.

5.3 Web Workers

Solutions in this category run third party code in Web Workers. Web Workers are a browser feature that was created to allow sites to create separate parallel execution environments. Each worker spawned by a site has its own JavaScript environment and can only communicate with its parent through the `PostMessage()` function. In addition, workers are only given a very restricted access to the shared browser API.

TreeHouse [5] provides a library that allows developers to execute code in Web Worker environments. Each worker runs a broker script that creates a virtual DOM for the code to interact with. The script listens for changes to the virtual DOM and if they are permitted on the real DOM, they are communicated to a script running on the main thread which then mirrors the changes onto it. Requests to a website's browser resources that are available to Web Workers are also mediated by the broker, but do not require any communication to the website.

6. Choosing an Approach

Each category of solution that has been suggested in the literature has the potential to meet all the requirements for this project. However, each is associated with its own pros and cons and based on these, it is possible to choose an approach that is most usable for developers.

Solutions using the Same Origin Policy and Web Workers both have similar pros and cons. Their main advantage is that they use mechanisms that browser vendors have built and maintain and their disadvantage is that they are restricted to using `PostMessage()` for communication. `PostMessage` allows JavaScript to send a `message` event between origins or JavaScript threads and to get a response, it must register a listener for a `message` and wait for the website to respond with a response `message`.

Solutions using a separate JavaScript interpreter have the advantage that they can be used to isolate any third party JavaScript, regardless of how it is written or what resources it requires. The biggest disadvantage of these solutions is the overhead of running a JavaScript interpreter in the browser. For example, in the js.js paper they found that in the Chrome browser most commands were slowed down a factor 100x to 200x.

Solutions using JavaScript subsets have the advantage that they allow third party code to execute at native speeds. This improvement in performance comes at the cost of reducing the set of commands they are able to evaluate. Since only a subset of the JavaScript standard qualifies as an object-capability language, there is the potential that some code written in regular JavaScript may include commands that do not exist in the subset.

In the end, the most usable approach for developers is to use JavaScript subsets. The SOP and Web Workers message-passing communication is too cumbersome to use and the performance overhead of running a separate JavaScript interpreter is too high. In practice, the disadvantage of the JavaScript subset approach does not severely impact its usability. The commands that are not possible in secure subsets of JavaScript are not often used, and most third party code has no problem working without them. On the rare occasion that incompatibilities do arise, it is usually a simple process to update the code to work in the environment.

7. Prototype Implementation

As a proof of concept, the JavaScript subset approach was used to isolate a common third party script for adding Google Analytics to a website. The prototype uses the JavaScript Secure EcmaScript (SES) library to isolate the Google Analytics script in an object-capability environment. Resources were then given to the environment, wrapped using the JavaScript `Proxy` API to validate all requests to them.

7.1 Secure EcmaScript

7.1.1 SES Lockdown

SES introduces the `lockdown()` function whose main purpose is to allow JavaScript code to share standard built-in objects without the risk of them being modified. This is possible using the `Object.freeze()` function provided to JavaScript through a standard built-in object. Once an object has been frozen, it can no longer be changed in any way.

7.1.2 SES Compartments

SES also introduces the `Compartment()` function whose main purpose is to allow code to be executed in an object-capability environment without access to unsafe global objects. The core JavaScript code that makes this possible is presented, simplified, in **Figure 1**.

```
const evaluateFactory = Function(`
    with (this) {
        return function() {
            'use strict';
            document; // bar
            (function () { return this.document }());  // error
            window.document; // error
        };
    }
`);

const object = {document: "bar"};
const proxy = new Proxy(object, {
    get: function (target, prop) {
        return target[prop]
    },
    has: function () {
        return true
    }
});

const evaluate = Reflect.apply(evaluateFactory, proxy, []);
evaluate()
```

Figure 1: evaluate JavaScript code in an object capability environment

The first feature to note is the `with` statement on line 2. The `with` statement takes an `expression` object (the `this` object here), and adds it to the scope chain for the evaluation of the following block. In the block, if JavaScript comes across an unqualified name, it looks for it first in the expression object. This is important because otherwise, JavaScript would look for the name through the scope chain until potentially finding it in the global scope which has references to unsafe global variables.

The second feature to note is the `Proxy` created on line 13. A `Proxy` takes an object and a handler, and intercepts requests to the object and handle them using custom logic. In this example, the handler will return any properties defined in the original object but will also claim to have any property asked of it. On line 22, the `Proxy` is used as the `this` object in the `evaluateFactory` function. As a result, when JavaScript evaluates the block, all unqualified names appear to be resolvable with the Proxy object and the global scope is never accessed.

The third feature to note is the 'use strict' string on line 4. This tells JavaScript that code after the expression should be executed in strict mode. In this case, the important feature of strict mode is that the `this` variable either references the object that called a function or is undefined. In non-strict JavaScript, `this` usually references the object that called a function but if that object doesn't exist (i.e. in an anonymous function) it will refer to the global object.

Putting it all together, it can be seen that this method prevents access to the security sensitive `document` object. It cannot be accessed using its global variable name as in line 5 (it gets intercepted by the `Proxy`), by trying to access it using the `this` in the anonymous function on line 6 (`this` is undefined in anonymous functions in strict mode), or by trying to access it through the `window` global variable (`window` is undefined since it is not implemented by the `Proxy`). In this example the `document` variable is bound to a simple string, but it can easily be replaced with an object that behaves exactly like its namesake, but with restricted capabilities.

7.1.3 Using SES

Figure 2 shows an example of how to use SES to evaluate third party code. Line 1 imports the SES module which adds the `lockdown` and `Compartment` functions to the website's global variables. Line 3 calls the `lockdown` function, freezing all of the standard built-in objects in the website's context. Line 5 creates an instance of the `Compartment` function whose environment has, along safe and now unchangeable standard built-in objects, the document and print variables added to its global scope. Finally, line 10 calls the `Compartment`'s evaluate function, which takes a string of JavaScript code and evaluates it in the `Compartment`'s environment.

```
1 import 'ses';
2
3 lockdown();
4
5 const c = new Compartment({
6   document: "foo",
7   print: console.log
```

```
 8 });
 9
10 c.evaluate(`print(document)`) // foo
```
Figure 2: using SES

7.2 Adding Resources

Normally, the Google Analytics script is fetched from a remote server during a website's runtime and executed in its global environment. The goal was to still fetch it at runtime, but evaluate it in a compartment environment with the principles of least privilege enforced. This task was accomplished in three stages. The first stage was figuring out the minimum set of browser resources the script needed to perform its task, the second stage was researching how the required resources could be safely given to the compartment environment without compromising its isolation, and the final stage was to actually build the prototype.

7.2.1 Finding Resources

The Google Analytics script is closure compiled. That means that though it may have been initially written by a human developer at some point, it has since been compiled into high-performance code. Closure compiled code behaves exactly the same as the original code, but unlike human written code, it is extremely challenging to read. This made the task of figuring out what resources the script needed more challenging.

The script was first evaluated in an compartment environment with no additional resources other than the safe standard built-in objects provided to it by default. As expected, it failed and threw an error. Based on the error message, it was possible to determine the object the script was trying to access. An example such error message is shown in **Figure 3**. A developer with knowledge of Web APIs can immediately recognize that the undefined object was supposed to be the `window` object, which has the `MutationObserver` function.

```
VM519:5638 Uncaught (in promise) TypeError: Cannot read property
'MutationObserver' of undefined
```
Figure 3: example Google Analytics script error due to missing resource

The next step was to add the missing object to the compartment environment and execute the script again. In the **Figure 3** example, rather than adding the entire window object to the environment, an object named window should be bound to the environment with only the MutationObserver property. This "debugging" process of executing the script and adding a missing object was repeated until the script no longer threw any errors.

Unfortunately, just because a script is no longer throwing errors does not mean it will actually work and the example in **Figure 4** illustrates why. Lines 1 and 2 simulate what happens during an iteration of the debugging process. Here, the resource variable is undefined and when the script attempts to access its foo property throws an error. Lines 3 and 4 simulate the next iteration. The resource variable is now set to what was missing in the last iteration. When the script attempts to access its bar property, though it does not exist, it no error is thrown. Thus, the

debugging process helps find all the global objects that are needed, but not all of their needed properties.

```
let window = undefined;
window.foo; // error
window = {foo: "bar"};
window.bar; // undefined
```

Figure 4: accessing an undefined object's property vs accessing an object's undefined property

To find all their properties needed by the script, the objects that were added to Compartment environment were wrapped using the Proxy API as in **Figure 5**. When the script executes with this wrapped object, all property accesses to it are logged.

```
const proxy = new Proxy(window, {
    get: function (target, prop) {
        console.log("getting: " + prop);
    }
});

const c = new Compartment({ window: proxy});
c.evaluate("googleAnalytics.js");
```

Figure 5: determining window object accesses

7.2.2 Safe Resources

Browser resources cannot be naively given to a compartment's environment. If care is not taken, a resource that allows the script to escape the environment may be given to it. An interesting example is shown in **Figure 6**. Here, a developer tries to create an SES compartment whose environment only has access to the `window.setTimeout()` function, a seemingly harmless function that executes code after a certain amount of time. Unbeknownst to them, they have accidentally compromised their compartment's isolation.

```
const safeWindow = {
    setTimeout: function() { return window.setTimeout }
};

const c = new window.Compartment({ window: safeWindow });
c.evaluate(`
    window.setTimeout(
        "window.alert('Hacked!')", 100
    )
`); // creates alert on screen
c.evaluate(`
    window.setTimeout(
        function() {
            this.window.alert('Hacked!'
        }, 100)
`); // creates alert on screen
```

Figure 6: unsafe resource

On line 1, the developer creates the `safeWindow` object whose only property is the `setTimeout` function and on line 5, they bind the object to the environment's global window variable. Line 6 shows the first way a malicious script can abuse this function. The developer had not done their research and did not know that `setTimeout` can take a string of JavaScript code, as its first parameter, that will always evaluate in the global environment. Line 11 shows another way this function can be abused. Here, a function is passed as the first parameter to `setTimeout`. A function A that is passed as a parameter to another function B, and is invoked in function B, is referred to as a callback function. When function A executes, its `this` value is set to the `this` value of function B. In this case, `setTimeout` is called by the global window object (line 2) and so `this` in function A references it.

If code executing compartment environment gets access to the `window` object, the compartment becomes completely useless. Through the `window` object any resource of the Web APIs can be accessed it could even be used to import a completely new script and execute it in the global environment. The `setTimeout` vulnerability is interesting because it was a particularly difficult one to catch but in reality, there are many other resources that need to used cautiously.

While building the prototype, every resource passed to the environment was researched to make sure it could not be used to compromise the isolation and if it did, it was wrapped to render it safe. For example, **Figure 7** shows how the `setTimeout` function was safely wrapped before being given to a compartment. Here, the actual `setTimeout` function is wrapped by another function that intercepts the `callBack` parameter and binds its `this` variable to the `safeWindow` object and in doing so, also ensures that `callBack` is actually a function.

```
1 const safeWindow = {
2     setTimeout: function (callBack, milliseconds) {
3         const safeCallBack = callBack.bind(safeWindow);
4         return window.setTimeout(safeCallBack, milliseconds)
5     },
6 };
```

Figure 7: safe resource

7.2.3 Building the Prototype

Once the resources needed for the Google Analytics script were known and any potential vulnerabilities in them found, it was finally possible to build the prototype.

Bibliography

[1] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In International World Wide Web Conference, 2009.

[2] Ter Louw, M., Ganesh, K. T., & Venkatakrishnan, V. N. (2010, August). AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium* (pp. 371-388).

[3] Agten, P., Van Acker, S., Brondsema, Y., Phung, P. H., Desmet, L., & Piessens, F. (2012, December). JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference* (pp. 1-10).

[4] Terrace, J., Beard, S. R., & Katta, N. P. K. (2012). JavaScript in JavaScript (js. js): Sandboxing third-party scripts. In *3rd {USENIX} Conference on Web Application Development (WebApps 12)* (pp. 95-100).

[5] Walfish, M. (2012). TreeHouse: JavaScript sandboxes to help web developers help themselves. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)* (pp. 153-164).

[6] Libert, T. (2018, April). An automated approach to auditing disclosure of third-party data collection in website privacy policies. In *Proceedings of the 2018 World Wide Web Conference* (pp. 207-216).

[7] Mulligan, D. K., & Perzanowski, A. K. (2007). The magnificence of the disaster: Reconstructing the Sony BMG rootkit incident. *Berkeley Tech. LJ*, *22*, 1157.

[8] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., & Lee, W. (2021). Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. NDSS.

[9] van Oorschot, P. C. (2020). *Computer Security and the Internet*. Springer International Publishing.