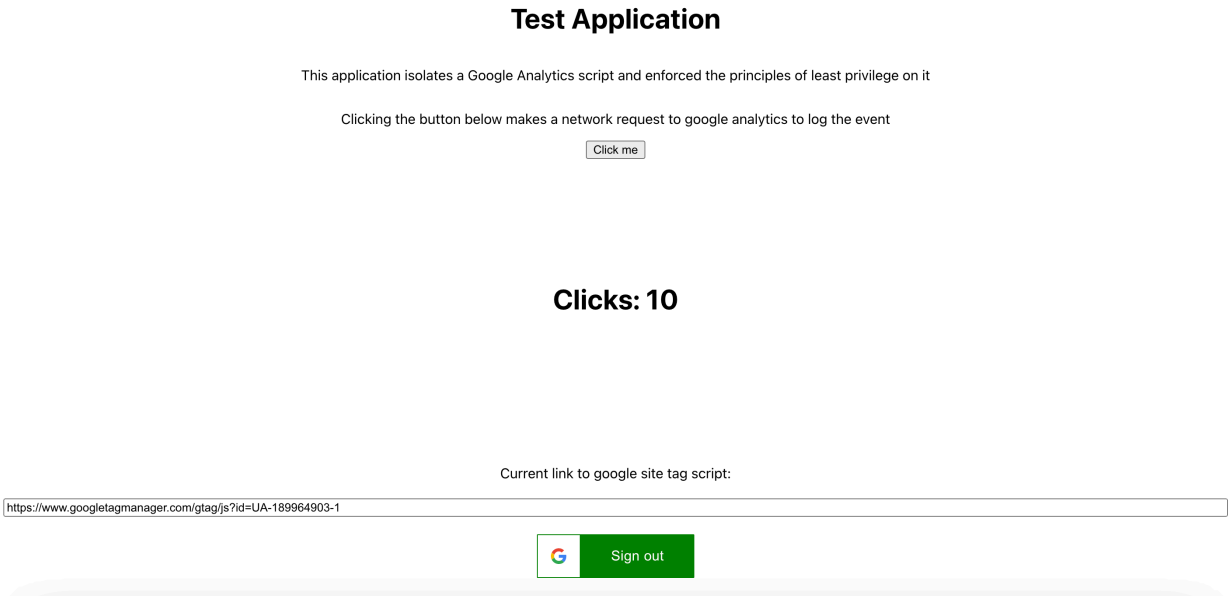
Isolating Third Party Code in a Website

Aiden Rourke



Abstract

Modern website development is highly dependent on third party code. Its use has allowed many website developers to punch above their weight, and build rich functionality in their websites that they could not have done by themselves. These advantages carry risks since third party code operates with the same privileges as the code written by the website's developers.

This project examines a method developers can protect their websites from untrustworthy third party code. It is able to mediate access to the most dangerous privileges, work in current and many legacy browsers, and is designed using academically accepted security principles.

This paper starts by exploring web technologies and the problem relating to third party code faced by website developers. It will then introduce the requirements for a potential solution aiming to solve the problem and past academic research in that area. Finally, it will introduce a real solution, implemented as a prototype, than can be used immediately to improve a website's security.

1. Introduction

JavaScript is a scripting language that allows website developers to implement complex features in their websites. 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 site redirections, and even manipulate objects shared throughout the website. These capabilities, useful for website developers, can be exploitable in the hands of a potential attacker. Unfortunately, the practice of including third party makes it drastically easier for an attacker to exploit them.

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 my 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 the only reason for the blind trust they give it is that 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 an end user's computer for the purpose of viewing and manipulating 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), JavaScript, and data such as images and videos. The browser takes these instructions and data, and uses them to display a 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, using them 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 shared thread, 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 access the browser resources given to the websites. 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 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 [9]. 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 should 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. The ability to mutate any, and access the dangerous of these resources should 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. Access to this category of resource should be mediated to prevent such an attack.

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 isolation mechanism must not depend on pre-processing third party code. If an isolation mechanism pre-processes code to remove restricted commands, it functions as a black-list since it defines what is not allowed. If an isolation 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 tried to solve it. The solutions that are relevant to the scope of this project can be divided into categories based on the isolation 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 isolate third party code from a website by executing 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 document inside of it. If the iframe's document has a different origin than the parent document, the browser treats the code running in their environments as being from different origins. 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.

5.2 JavaScript Subsets

The Web APIs and standard built-in objects are shared with code running on a site as global objects. To sandbox third party code executed in the same environment as the rest of the website, it must be prevented from accessing these objects. One way this can be achieved is by using an object-capability environment in which executing code cannot access an object without an explicit 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 that they 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 provided a reference to any of the global objects provided by the browser and references to necessary browser resources can be provided to code executed by the library.

JSand uses the JavaScript `Proxy` API to wrap browser resources before passing them to isolated code. A wrapper interposes between a caller and a resource, and is capable of validating requests using 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 provided with 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 does not have any access to the Web APIs but the Js.js library provides an API to allow developers to bind objects into the isolated environment. This makes it possible to provide wrapped browser resources to the environment, capable of validating requests from the environment code.

5.3 Web Workers

Solutions in this category execute 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 initialized with 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 type of solution that has been suggested in the literature has the potential to meet most of the requirements for this project. However, each is defined by an advantage and disadvantage. Based on those, it is possible to choose an approach that is best for developers.

Solutions using the Same Origin Policy and Web Workers can be grouped together in terms of their advantage and disadvantage. Their advantage is that they make use of a time tested and browser supported isolation mechanisms. Their disadvantage is that they are restricted to using `PostMessage` API for communication. The problem with the `PostMessage` method of communication is that it is asynchronous. To provide access to some browser resources, developers must either change isolated code to handle asynchronous requests or if possible, intercept requests and synchronously return a faked result.

Solutions using a separate JavaScript interpreter have the advantage that they can give full execution control over code executing in their environment. Infinite loops and memory abuses can be prevented using checks in the interpreter loop. The disadvantage of these solutions is the performance overhead due to executing commands in a JavaScript interpreter running on the browser's. For example, in the js.js paper, they found that in the Chrome browser, their JavaScript interpreter slowed down most commands by a factor 100% to 200%.

Solutions using JavaScript subsets have the advantage that they execute third party code in the same environment as the rest of the website. Thus, JavaScript commands in the environment perform at the same speed as outside of it. Their disadvantage is that the set of commands they are able to evaluate is reduced. 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.

Based on my own experience as a web developer, it is impossible to imagine my colleagues running third party scripts in Web Workers or iFrames, constantly having to edit the scripts to handle asynchronous communication. It is equally impossible to image them being willing to add a heavy JavaScript interpreter onto their websites. The only approach I can actually imagine them using is the JavaScript subset approach. In practice, its disadvantages do not severely impact its usability. The commands that do not exist in secure subsets of JavaScript are not commonly used, and on the rare occasion that incompatibilities do arise it is usually easy to update the code to make it work.

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, which was then provided the minimum set of access controlled resources to allow it to perform its function.

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. 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 the interpreter comes across an unqualified name, it first looks for it 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 processes them using the handler. 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 referenced.

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 `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 binds the `lockdown` and `Compartment` functions to the website's global environment. Line 3 calls the `lockdown` function, freezing all of the standard built-in objects in the global environment. 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 in 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. To do so, the minimum set of browser resources the script needed to perform its task had to be determined, and each required resource had to be researched to determine if or how it could be safely endowed to the compartment environment without compromising its isolation.

7.2.1 Finding required resources

The Google Analytics script is closure compiled. That means that though it may have been initially written by a human developer, 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 difficult.

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 resource the script was trying to access. An example of such an error 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` property.

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

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

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

exist, 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 created 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 Adding Resources

A compartment's environment cannot be naively endowed with browser resources. If care is not taken, a resource that allows the script to escape the environment may be provided 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 environment'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 its callback references it.

If code executing in a compartment environment gets access to the `window` object, the isolation is completely compromised. 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 resources that need to used cautiously.

While building the prototype, every resource passed to the environment was investigated to make sure it could not be used compromise the isolation and if it did, a way was found to make it safe. For example, **Figure 7** shows how the `setTimeout` function can be secured before being passed 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: making a resource safe

7.3 Building the Prototype

Once the resources needed for the Google Analytics script were known, and any potential vulnerabilities with them found, it was finally possible to build the prototype. The implementation consisted in turning the necessary resources into access controlled resources, and a script to create an SES compartment with them and evaluate the Google Analytics script inside of it.

7.3.1 Access Controlled Resources

Access controlled resources, also known as a secure resources, are JavaScript objects that act as an access control mechanism for a particular resource. A diagram for how they handle a request to a resource is shown in **Figure 8**. It begins with a script making a request to a secure

resource. The request is handled by a resource wrapper, which checks a security policy to determine if the request is allowed or not. If it is allowed, the wrapper handles the request to the resource.

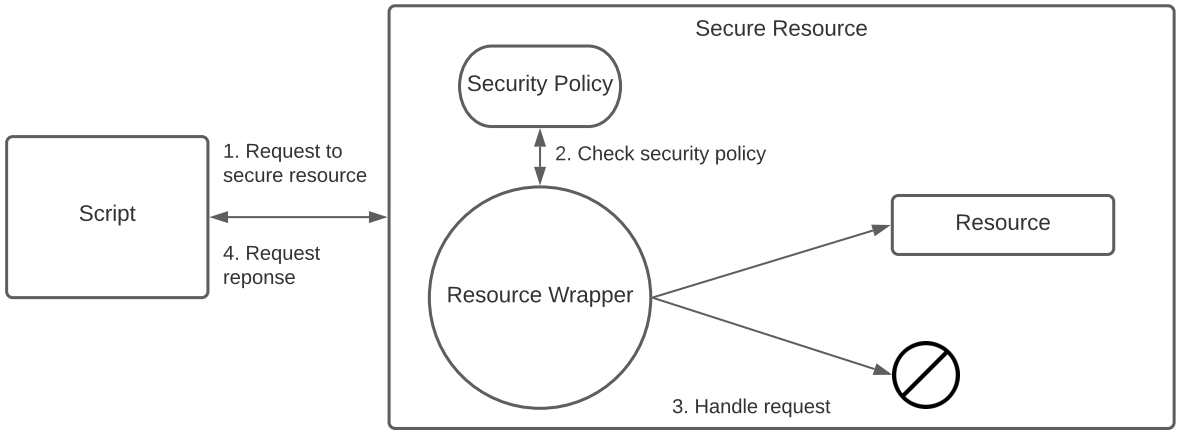


Figure 8: requests to a wrapped resource

Security policies are simple JavaScript objects that define what third party code is allowed to request of a resource. A simple example is shown in **Figure 9**. Here, the security policy is stating that `get` requests are allowed to the secure resource's `setTimeout` property. Security policies act as a white-lists, all requests that are not explicitly allowed by it are denied.

```
const windowPolicy = {
    get: {
        setTimeout: true
    }
};
```

Figure 9: example of a security policy

Resource wrappers are JavaScript `Proxy` objects that handle all requests to a secure resource. On top of validating requests, they are responsible for ensuring that none of the requests can be used to compromise an environment's isolation. **Figure 10** shows an example of how a resource wrapper safely handles requests for the dangerous `setTimeout` function. When a request for it is made, the wrapper ensures that it is returned as a secure resource.

```
const timeout = secureSetTimeout(window.setTimeout.bind(window));
const windowWrapper = new Proxy(window, {
    get: function (target, prop) {
        if (windowPolicy.get[prop]) {
            if (prop === "setTimeout") {
                return timeout;
            }
            return target[prop];
        }
    }
```

```
11 });
```
Figure 10: example resource wrapper

Figure 11 shows how a wrapped resource and security policy are combined to create a secure resource. The result of a function like this, if done correctly, can be safely provided to a compartment environment.

```
1 function secureResource(resource, securityPolicy) {
2     return new Proxy(resource, {
3         get: function (target, prop) {
4             if (securityPolicy.get[prop]) {
5                 // vulnerability conditions
6                 return target[prop];
7             }
8         }
9     });
10 };
```
Figure 11: simple secure resource

7.3.2 Evaluating the Script

After secure resources were created for every necessary resource, the Google Analytics script could be safely evaluated. In **Figure 12,** the `window` variable is accessed the same way inside and outside of the compartment but in the compartment, it is handled by `secureWindow`. Since the resources could be provided in this way, there was no need to edit the Google Analytics script. All that needed to be done at this point was create a secure instance of each required resource, add them to the compartment, and evaluate the Google Analytics script inside of it.

```
1 const secureWindow = secureWindow(window, windowPolicy);
2
3 const c = new window.Compartment({ window: secureWindow });
4
5 window.setTimeout(cb, i); // Handled by global window
6
7 c.evaluate(`
8     window.setTimeout(cb, i); // Handled by secureWindow
9 `);
```
Figure 12: seamless security

8. Results

To meet the requirements of this project, a solution had to be found that allows developers to enforce least privilege on third party code's access to security sensitive browser resources, is compatible with current and legacy browser technologies, and operates as a white list mechanism.

The security sensitive browser resources were divided into three categories: Web APIs, standard built-in objects, and the JavaScript engine. The prototype implementation was able to mediate access to the first category of resources, the Web APIs. The object-capability environment provided by SES allowed the prototype to create fine grained privileges for access to these resources. For the second category, the standard built-in objects, the prototype was able enforce least privilege on unsafe objects that can be used for side-channel-attacks since SES does not give a compartment access these dangerous objects by default. Additionally, the SES lockdown function completely blocks a third party script's ability to mutate standard built-in objects which nullifies their threat. The SES library cannot mediate access to the final category of resources, the JavaScript engine. An object-capability environment can only control the objects available to code and not its actual execution. In sum, the prototype built in this project did not completely meet the requirement to be able to enforce least-privilege on all security sensitive browser resources, but it was able to meet it for the most dangerous ones.

The technologies used by the prototype work in all the recent browsers. There are some older browsers that do not support "`strict mode`" but they are only used by less than 1% of users. It is also likely that browsers will continue to support this feature and there are no plans to phase it out. Thus, the technologies used by the prototype allow it to meet the requirement that it can be used immediately, and with reasonable certainty that it will work on the vast majority of users browsers.

The object-capability environment created in an SES compartment denies access to all global objects by default. Even the unchangeable standard built-in objects the compartment shares with the main site have to be programmatically added to the compartment by the library. Every additional object must be added by the application creating the environment. This default-deny approach used by the prototype meets the requirement that the isolation mechanism function as a white-list rather than a black-list.

9. Conclusion

This project discussed some of the approaches that developers can take to secure their websites despite the inclusion of third party code. It also implemented a prototype using one of the approaches to show how it can be done. The prototype met most of the important requirements, but there is still room for improvement. Further work in relation to the prototype could be to automate the process of finding a script's required resources and to automate the patching of vulnerabilities in the required resources. Having either or both of these features would drastically reduce the time needed for developers to use the approach themselves.

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.