Same Origin Method Execution (SOME) Exploiting A Callback for Same Origin Policy Bypass

Ben Hayak
Twitter: @BenHayak
Initial submission date (Nov 5, 2014)

Technical Document submitted to
BlackHat UBM conference

Nov 2014

Date of publication, May 2015

Abstract

As time passes, the Internet grows tremendously and results in ever-increasing amount of our most private data being stored online. Simultaneously, many companies encourage us to use their online services, such as online banking, shopping, medical services, social media networks, and, most importantly, various cloud storage services. These are capable of maintaining a backup of every piece of information we have, often without us being aware of the scale of the process. In order to have an interface between the costumers (us) and the aforementioned services, the companies create Web Applications.

When banking, storage and other services are offered though a web application interface, personal sensitive data, along with other procedures like financial instructions, are exposed to a multitude of security risks. Accessing these services, the browsers are responsible for requesting and rendering data and instructions to and from the web servers. This essentially makes the users dependant on the security standards of these two components, namely the service provider (i.e. the web application developers) and the browser's security mechanisms.

To satisfy our need for a faster and more efficient user experience while using web applications, new protocols and techniques are formulated. They supply the web application developers with a powerful variety of options. However, it must be kept in mind that "with great power comes great responsibility". This means that diversity and power can yield disastrously wrong implementations, which expose users to a plethora of unexpected attacks.

One of those wrong implementation resulted in a discovery of a new web application attack called "Same Origin Method Execution" (SOME). Importantly, the implications of the attack envelop a risk level close to one of the most dangerous client-side web application attacks ever to be documented, namely Cross Site Scripting. The client-side web application attacks often aim to abuse a user session on a vulnerable trusted website by forging an arbitrary link or a web page. This is done in the goal of forcing the user's browser to execute malicious actions of the vulnerable trusted website. Until the present day, we have witnessed a range of client-side web application attacks, such as XSS, CSRF, an so on, which exposed users to threats in web applications varying in impact. With the flexibility of SOME, a successful exploitation can substantially elevate the level of providing an extremely critical control over a user's session and, in some cases, it may even compromise the vulnerable site.

This paper is dedicated to an in-depth discussion of the Same Origin Method Execution. It delineates what SOME actually is, and explores which instances are vulnerable to it and why. Furthermore, the paper highlights different techniques of exploitation,

the obstacles during exploitation stages, as well as means to overcome the barriers. Ultimately, an ideal defense approach to the attack mitigation is introduced. Crucially, it gives a glimpse into a novel defense javascript library that the author is currently working on.

Contents

1 Introduction

Same Origin Method Execution (SOME) is a web application attack that allows hijacking the execution of Web-Application "Document-Object-Module" and/or scripting methods on behalf of users. In the SOME attack, the victim initially visits a malicious link or is lured by a malicious advertisement. Subsequently, an unlimited predefined set of actions is executed by the victim's user agent (as in an XSS attack). By abusing the victim's session, SOME can perform actions exactly as if the victim has triggered them on his or her own. Unlike many other similar attacks, SOME neither requires tricking the user into clicking on hidden objects, nor is confined in terms of user interaction, browser brand, frame busting, HTTP X-FRAME-OPTIONS/Other response headers or a particular web-page. In fact, when a web-page is found vulnerable to SOME, the entire domain becomes exposed to its resulting vulnerabilities.

1.1 Thesis Outline

This thesis is divided into five main parts. Following directly after this Introduction, Part one, "Same Origin Method Execution" (Chapter 2), provides an overview of how web security mechanisms are bypassed by the attack. It presents the conceptualization behind the attack and describes the legitimate behavior that is abused by it. Following the conceptual outline, Section 2.2 showcases the details behind the vulnerability. It ponders a question of which implementations are vulnerable to SOME and highlights a plethora of reasons causing this particular weakness. Later sections elaborate on the attack flow and describe a basic "Proof of Concept" exploit, while depicting the status of Same Origin Policy during the process as well. Finally, Section 2.6 introduces a technique of hijacking a web functionality of a trusted site. It is followed by a short summary and conclusions that can be drawn up to this point.

The third chapter "SOME Advanced Aspects" is dedicated to introducing the advanced aspects of the attack. It emphasizes noteworthy possibilities for exposing vulnerable instances and tackles extended risks, such as examples taken from real-life vulnerability-centered scenarios.

The forth chapter, called "Mitigation and Bypass", focuses on relevant browser and web security aspects. Mitigation mechanisms take a center-stage and we wonder what can make them a proper obstacle during the various exploitation stages. Following an overview of each and every obstacle, the chapter either presents a bypass, or a complete

subversion technique pertaining to it.

The fifth chapter entitled "Ideal Defense Approaches" introduces research conclusions originating from vendor fixes and some creative mitigation approaches. The chapter discusses active solutions, including source code examples of an existing prototype capable of mitigating SOME-based exploits. This chapter particularly showcases a peak in the area of client-side mitigation technique (currently in its development stages). It seeks to mitigate SOME vulnerabilities in callback implementations, while still maintaining high flexibility and performance.

The salient five chapters outlined above are accompanied by the sixth and final component, which can be consulted for a list of figures and listings. It also contains acknowledgements and curriculum vitae of the author.

2 Same Origin Method Execution

The following pages will be dedicated to an overview of the attack and a wide-ranging discussion of impact that Same Origin Method Execution has. The argument emphasizes the threat and implications, while the later sections expand on issues of detection and exploitation of the attack. Furthermore, a detailed characterization of the particularities of SOME will be mapped out. Consequently, this discussion will introduce the concept and propose a novel technique of exploiting wrong callback implementations (SOME exploitation). Note that sections other than Section 2.2 deliberately presume a vulnerable callback URL was found and elaborate on how to exploit these specific scenarios.

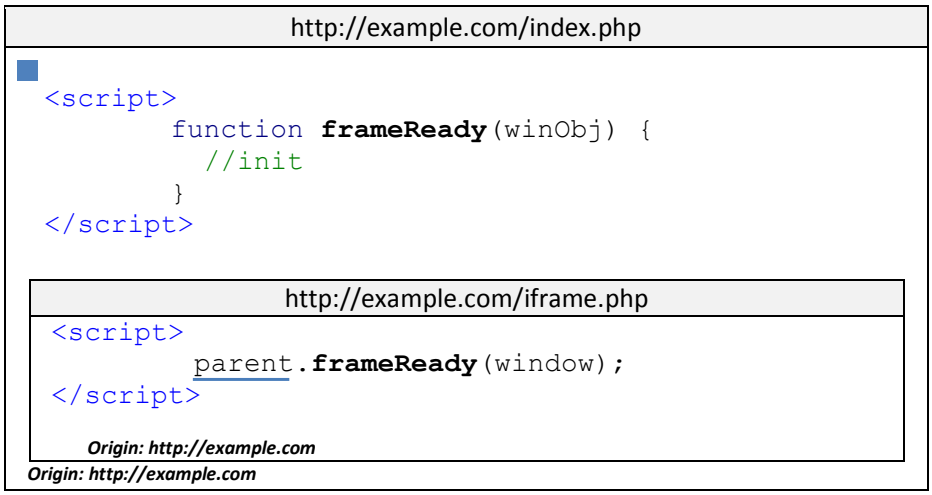
2.1 Introduction and Overview

SOME (Same Origin Method Execution) is an attack that takes advantage of SOP (Same Origin Policy)[1] rule by abusing the concept of callbacks. It does so in order to execute scripting methods across various windows/contexts of the same-origin, thus executing an arbitrary flow of unwanted actions on a trusted site on behalf of users as a consequence.

Same Origin Policy mechanism will allow an evaluation of a script from one or the other document belonging to the same origin (e.g. domain, protocol, port). Due to said same origin, the Same Origin Policy will not restrict access to either document's DOM. In other words, a web-page can access properties and evaluate methods belonging to the DOM of a different page of the same origin.

[1] MDN, *Same-Origin-Policy*, `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`

Figure 2.1: A child window executes a method handler of its parent document:



Shown in figure 2.1:

- The framed document (iframe.php) uses its "parent" reference to **execute a method** of another window to notify about a "ready" event.
- As a result, Same Origin Policy allows the execution of the script since it was originated from a document of the same origin.

SOME attack abuses this germane SOP rule and the callback concept by moving the execution context from the original endpoint document (callback URL) towards the desired/target document/s of the same origin. To achieve this, the attack must begin with a setup in the attacker's website, which would entail forging an attack surface consisting windows/frames. Following the setup, the attack is evolved into redirecting the forged documents' locations to the designated target/s, and next to a malformed callback URL accordingly. Then the attack ends by tricking the user agent into executing arbitrary methods at the context of a chosen target document. To control the flow and forge a successful attack, SOME setup has to be designed in a way that it keeps an access reference between the window objects at all times.

By employing SOME attack one can assemble an arbitrary callback URL hosted on a trusted origin to execute client-side scripting methods in the context of any target document of the same origin/domain. As much as it might sound similar to Cross Site Scripting (XSS), most of the dynamic callback implementations (example to be found on the listing 2.1) restrict any patterns other than alphanumeric and few other characters (usually underscore: '_' and a dot: '.' example regex may be: "[A-Za-z0-9_.]"). Unless the latter is combined with additional injections, it basically eliminates XSS vulnerabilities. Despite this extensive restriction, with SOME it is possible to hijack execution

of any method that does not require arguments with neither any user-interaction nor the user's knowledge. Such method execution includes: clicks, form submissions, form input value tampering, JavaScript functions and similar (e.g. element.click(), privateForm.submit(), inputElement.stepUp/stepDown(), element.select(), element.focus(), JsDefinedFunction(), jQueryFunc() and so on).

All in all it boils down to the fact that the Same Origin Method Execution takes advantage of the nature of user agents and brings in the possibility of hijacking client-side method execution by a sole usage of alphanumeric characters and a dot. The impact of the attack is two-fold. On the one hand, its ramifications are bound to the capabilities exposed by the targeted vulnerable web application (method execution of **any page** hosted in the vulnerable domain can be hijacked). On the other hand, the impact is also tied to the role that a user plays. By attacking ordinary users, the attack may lead to sensitive information leakage, which can signify revealing private photos, transfer of funds, item purchases, and even a complete account takeover (when attacking OAuth services). If the target user is an administrator, a SOME attack may also compromise the entire web application.

2.2 The Vulnerability - False Callback Implementations

As this chapter will focus mostly on technical exploitation details, it is important to first understand how callback implementations are vulnerable and why. Further, it is crucial to foreground and foster understanding of the motivation behind developing a potentially vulnerable callback endpoint.

User agents enforce access control mechanisms, such as Same Origin Policy. The latter provides restricted access to and from cross origin documents. Occasionally, web application services functionality requires some way for overcoming this restriction. For instance, it is quite common that a narrow communication channel between two or more controlled sources (i.e. origins or web applications) is allowed. For that purpose, browsers offer various options, such as Cross-origin resource sharing (CORS), Cross-window Messaging (postMessage) JSONP[2], and the callback concept, to name just a few examples.

Zooming in on the Callback, it is a computer programming concept which allows passing a name of a function to a service in order to inform that service how to "call back" (execute the given function) when an event has occurred.

Detecting whether or not a specific callback URL is vulnerable to SOME is quite simple. In essence, any HTTP request that leads the document into arbitrary function executions puts the entire domain at considerable risk. Such callback instances may be implemented

[2]JSONP is also considered a variation of the callback concept

in web applications' documents, servers, or plug-in systems on which they rely on, thus exploiting any of these will initiate callback execution under the premise and permissions of a trusted domain.

The most common web-based callback implementations to date are JSONP APIs[3], and Flash applets. Both JSONP APIs and Adobe Flash applets commonly use the concept of callback to overcome SOP and/or inform another browsing context about a certain event. This could include, for example, information about pending data from a different source origin or an external status change (e.g. a ready event following a flash applet's loading completion).

Many callback implementations provide "friendly" callback URLs with intuitive parameter names:

1. http://trusted-site.com/callback-endpoint?**callback**=callback_function_name
2. http://trusted-site.com/callback-endpoint?**cb**=callback_function_name
3. http://trusted-site.com/callback-endpoint?**jsonp**=callback_function_name
4. http://trusted-site.com/callback-endpoint?**cmd**=callback_function_name
5. http://trusted-site.com/callback-endpoint?**readyFunction**=callback_function_name

There are different variations of implementing this concept, though a vulnerable instance is an instance that yields the user agent's interpreter towards a successful evaluation of a controlled function name under the context of a trusted site.

In the following, one can see a selection of vulnerable examples taken from real-life cases and scenarios:

Listing 2.1: Example markup of a vulnerable callback implementation with an attempt to transfer JSON data via a dynamic callback (Google Plus)

```
1 URL: http://trusted-site.com/callback-endpoint?cb=callback_function_name
2
3 <html><body><script type="text/javascript">
4         window.opener.callback_function_name({'status':0,'token':'ItHumYWl
              [...snip...]uDJNuXmAI','oauthstate':'1bg5a0LEE[...snip...]
              cgMDo','tokenid':'a11b11a7a6537222','tokenexp':'0','gid
              ':'4028278272713915914','siteid':'6','displayname':'James Bond
              ','profileurl':'http://profile-url.com'})
5       </script></body></html>
```

Listing 2.2: Example Flash applet claimed to be used on over 100,000 websites. The applet executes a given javascript callback function upon loading complete based on a FlashVars parameter (video-js.swf plugin)

[3] wikipedia, *JSONP*, http://en.wikipedia.org/wiki/JSONP

```
URL: http://trusted-site.com/callback-endpoint.swf?readyFunction=
    callback_function_name

if (loaderInfo.parameters.readyFunction != undefined) {
    ExternalInterface.call(_app.model.cleanEIString(loader
        Info.parameters.readyFunction), ExternalInterface.objectID);
}

[...snip...]

public function cleanEIString(arg1: String): String {
    return arg1.replace(new RegExp("[^A-Za-z0-9_.]", "gi"), "");
}
```

The key point in both vulnerable cases is that they allow using a dot in the payload. Therefore, they both provide a dangerous external control over the callback function name that gets evaluated by the user agent. Prior to the fix, this in fact opened the entire Google Plus domain to SOME. Similarly, every domain that used video-js plugin (or still uses an older version of it) became affected by Same Origin Method Execution issue.

2.2.1 Motivation Behind Developing Prone Callback Endpoints

We have covered the technical details that make for a vulnerable callback instance. Similarly, snippets taken from real vulnerable examples have been presented. However, especially since there are other alternatives, it is important to understand why would developers implement faulty callback endpoints in the first place.

An answer to this question put forward here is based on solid conclusions from statistical research. The data was collected from top sites on the web and relies upon the type of implementation.

2.2.1.1 Plug-in Systems

In case of plug-in systems, such as Adobe Flash, the answer is quite simple. It predominantly stems from the fact that whenever an applet is embedded in a web application, it has its own distinct context. For making a more interactive and smarter applets, developers may want their applets and embedding documents to react and respond to certain events. For applets, a currently most common technique for notifying the embedding document about certain events (e.g. loading complete, status) integrates executing JavaScript methods[4]. To create an even greater flexibility (e.g. plug-ins that serve large-scale web applications) developers externalize control via FlashVars[5]. That way the embedding document can influence how is it possible for the applet to call it back.

[4] Adobe, *Accessing JavaScript functions*, http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf626ae-7fe8.html

[5] Adobe, *Pass variables to SWFs | FlashVars*, https://helpx.adobe.com/flash/kb/pass-variables-swfs-flashvars.html

Granting this control (e.g. listing 2.2) may expose the entire domain on which the applet is hosted on to XSS and/or SOME vulnerabilities.

2.2.1.2 Callback Endpoint Documents

In case of callback endpoint documents' implementations it is all mostly about the developers' habits and preferences for deploying the simplest solution aimed at overcoming the SOP restrictions. Random reasons behind giving an external control over callback execution will make a vulnerable instance. As it stands, the motivation behind building a vulnerable implementation for the majority of the instances consists of the following:

- The service requires "secure delegated access" to third-party server resources on behalf of a resource owner (**OAuth**).
- The service is not willing to lose the currently present content and thereby opens a **pop-up** window.
- It is **simple** and requires less effort in comparison to the alternatives.
- Lack of security **awareness**.

Honing on the issue in a holistic way leads one to see that when a developer aspires to build a service that should perform an operation of pulling user data from a third-party domain, the service would need the resource owner (the user) to **interact** for approval. The behavior affects, for example, importing contacts, friend-lists, phone-books, supporting "Login with" tokens and so forth. The most common technique tasked with fulfilling the above described need is OAuth. In order to gain access to third-party resources using OAuth, the service shall utilize a third-party endpoint (OAuth dialog) that will ask for the resource owner's approval. The problem with this process is that redirecting the service to an OAuth dialog means losing the content of the currently open service document. For overcoming this problem, developers open a pop-up window to display the dialog in a singular browsing context. Once the user permits or denies access to the service, the OAuth dialog pop-up will be redirected to render a callback endpoint hosted on the service domain. This document should eventually notify the service that the process has been completed.

For the new pop-up window to notify the service window upon approval, denial or for it to transfer access tokens or similar data, developers may implement callback endpoints that use a script referencing the "opener" window for executing a callback method of the service. When developers also opted for providing the service with the decision on how to "call it back" through a callback parameter, the entire domain becomes vulnerable to SOME.

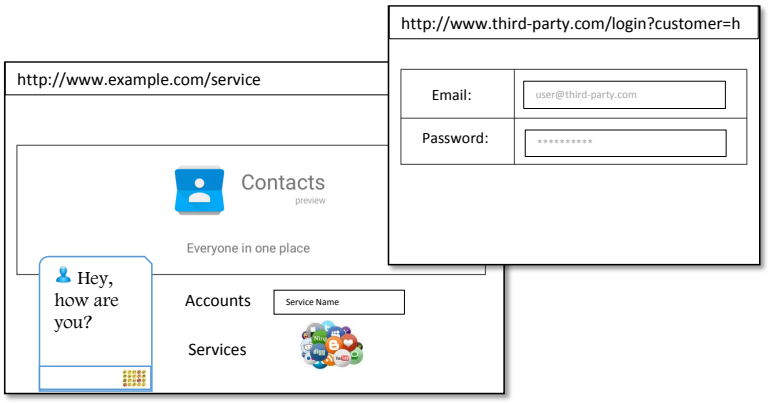


Figure 2.2: Requesting third-party resource access without losing the content of the currently open document.

Executing a callback method whilst preserving the currently open document content can be achieved by using several solutions such as Cross-Domain Messaging (PostMessage). However as opposed to the aforementioned technique, doing so requires additional efforts of various composite verification and security checks.

The reasons delineated above explain the lack of awareness regarding security risks such as SOME. They also provide an understandable rationale of why the developers may build prone callback endpoints.

2.3 SOME Attack Flow

The next sections will describe the actual attack flow and discuss the most basic aspects of the attack with an ideal use case. In turn, the later sections supply a step-by-step technical outline regarding how a basic attack of only a single action hijacking might be initiated. Consequently, this part will demonstrate how SOME can be utilized to "turn off every CSRF protection". In addition, for the attack to work and be capable of executing methods or stealing private information on behalf of the target user's identity, no interaction from the side of the user-victim is required. The following sections will not elaborate further on real case exploitation nor on how to hijack a flow of multiple actions as such will be covered in Chapter 3 entitled "SOME Advanced Aspects".

It is important to first understand the general attack flow:

1. The target user agent follows an arbitrary link.
2. This arbitrary link causes the user agent to create windows/frames and to subsequently follow a set of redirection.

3. The user agent navigates one of the windows to render the target document and others to render the vulnerable callback URL, both hosted on the same trusted origin.

4. A client side method is executed by the user agent in the context of the trusted site. This results in a hijacking of web actions under the targeted user's session, as if the user triggered these actions him or herself.

2.4 Creating Same Origin Environment

Setting up Same Origin Method Execution begins almost from the very last step, namely the targeted document (of a trusted site). The target document must be set to a page that contains the action that the attack aspires to hijack. After abusing a vulnerable callback URL, the attack ends with an execution of a scripting/DOM method. In order to execute a valuable method on behalf of a targeted user, a SOME attack must designate the execution context to the target document. Hence the attacker must create a reference between the vulnerable callback endpoint and the target document.

There are several adequate options for crafting references between window objects such as, for example, creating frame elements. Although considering the nature of web browsers [6], nowadays the following options are found to be the best for conducting the most efficient SOME attack by the author:

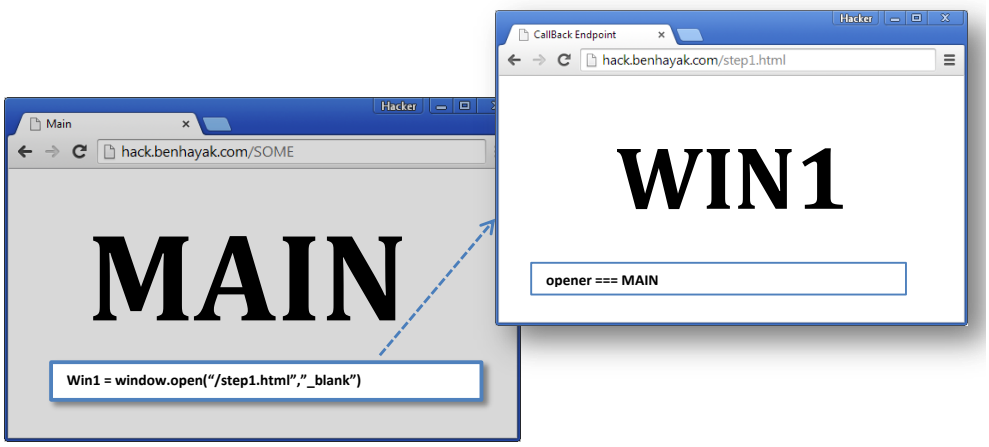


Figure 2.3: Creating the environment

Figure 2.3 above describes the following:

1. A user agent follows an arbitrary link which leads to a (main) website. In turn, the website launches the attack.

[6] W3C org, *Accessing other browsing contexts*, `http://www.w3.org/TR/html/browsers.html#accessing-other-browsing-contexts` (Oct 2014)

2. The main document uses javascript code to open a new browsing context window.

3. A reference is then added to each of the window objects.
   a) A reference to the new window (i.e. "WIN1") is returned from the *window.open* function and saved as the variable "Win1" in "MAIN".
   b) A reference to the main window (i.e. "MAIN") is added as the window "opener" property in "WIN1".

Since all of the described documents stem from the same origin, each of the browsing contexts may use this reference to access and set properties of the others' DOM.

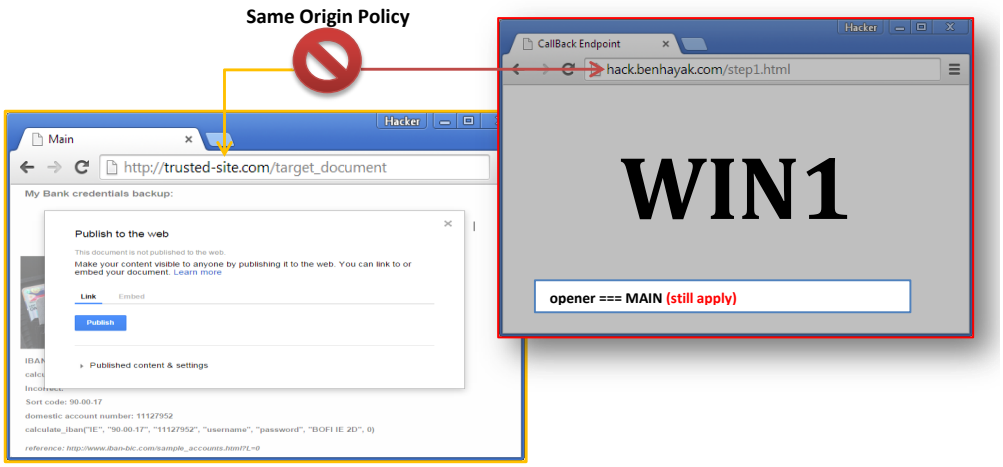
2.5 Designating the Executing Context

When a proper environment is ready, a SOME attack must specify a target document chosen from any of the trusted site's documents. The chosen document shall contain the web functionality that the attacker aspires to hijack. At the same time, in order to hijack the desired functionality, the exploit must predict the reference to it in the target DOM (this behavior is detailed further in the later section 2.6.1).

2.5.1 Designating a Target Document

Once the environment is ready, designating a target document is not particularly hard as the main window (the "opener" window) can simply be set to any chosen target document URL via redirection.

Figure 2.4: The environment after the redirection:



In the figure 2.4, the main window document was redirected to render the content of a trusted website's (target) document.

The following piece of code provides one example within a multitude of various redirection options. They may all be used for the purpose of designating a target document:

```
URL: http://hack.benhayak.com/SOME

<script>
  function start_SOME() {
    document.location = "http://trusted-site.com/target_document";
  }
  win1.onload = start_SOME();
</script>
```

Listing 2.3: Optional JavaScript redirection approach (intended to execute in "main" window document)

Note: The target document may be set to any webpage hosted on a trusted site. To facilitate an understanding of the concept, the example describes a theoretic trusted document which includes the user's private bank credentials information, as well as a DOM containing a "Publish" button (common on cloud backup service implementations).

Following the redirection, the main window object remains at its original allocated memory address, therefore the "opener" reference of "WIN1" still points to it. In other words, it is possible to use the "opener" reference of "WIN1" to designate the execution context to the context of the "MAIN" window ("WIN1" may execute the following: *opener.publish();*), even after the redirection took place. However, as soon as the redirection occurs, the "Same Origin Environment" is broken. Consequently, if user agents were to attempt an evaluation of cross origin instructions, they would detect an SOP violation and thwart execution with an error similar to:

Blocked a frame with origin "http://hack.benhayak.com" from accessing a cross-origin frame.

After completing the steps of designating the target document via the aforementioned redirection, one may proceed to creating a setup that ensures the attack's success. This should be as follows:

1. Wait for the target document to complete loading. This could be done by setting a timeout in "WIN1" prior to taking further operations, for example:

```
URL: http://hack.benhayak.com/step1.html

<script>
  //Wait 3 seconds for the trusted-site's DOM loading completion
  setTimeout(function() {/*forge-callback-execution*/} ,3000)
```

```
6 </script>
```

Listing 2.4: An Optional JavaScript approach of waiting for a cross-origin document to complete loading (intended to execute in "WIN1" window document)

(This step is essential for pinpointing object references successfully mentioned in the section to follow 2.6.1 “Predicting the Object Reference within the Target DOM”)

2. **Reconstruct a "Same Origin Environment"** to regain a full access to the cross-browsing context DOM again (additional redirection).

Fulfilling these two steps will lead to "method execution" in the context of a designated target document.

2.5.2 Method (callback) Execution

A vulnerable callback endpoint hosted on a trusted site will normally lead to an execution of a callback function under an explicit browsing execution context, in accordance with a given HTTP parameter (see section 2.2 entitled “The Vulnerability - False Callback Implementations”). Since callback endpoints usually contain only a narrow set of instructions designed for serving a specific objective, hijacking arbitrary method execution under the endpoint’s context would usually have harmless effects. In order to elevate impact and risk, the attack will utilize the aforementioned setup to replace the execution context with the use of a window reference.

At the stage where the "MAIN" window is set to render a target document, the reference is ready to be used. The exploit can include the "opener" reference as part of the callback URL parameter value, then use a redirection again to navigate the new browsing context ("WIN1") to that arbitrary URL. This will both reconstruct a "Same Origin Environment" for overcoming SOP restrictions, and delegate the execution context to the "opener" ("MAIN") window (which was set up in the aforementioned section 2.4). This will consequently expand the possibilities of executing further DOM objects’ methods of a chosen target document (i.e. an expansion of hijacking possibilities).

Figure 2.5: Abusing a vulnerable callback endpoint to execute an alert in the context of its "opener" window

Figure 2.5 shows post-redirection Same Origin Method ("alert") Execution in the context of the target document window ("MAIN") triggered by a script in "WIN1". This is the result of:

1. Crafting the vulnerable callback URL by setting the callback parameter as follows:
   a) Replace the execution browsing context with the context of the designated target document window - referenced as "opener" property of "WIN1" (i.e. callback=opener.X.Y.Z).
   b) Set the desired method ("alert") of the target document's DOM (i.e. callback=opener.alert).
2. Reconstructing a "Same Origin Environment" by **redirecting** the document of "WIN1" to the formerly crafted arbitrary callback URL.

Ultimately, a redirect to the crafted URL will manipulate the vulnerable callback endpoint into hijacking an "alert" method execution within the context of the designated target document.

2.6 Hijacking Arbitrary Method Execution

As in many other client-side web vulnerabilities, a SOME exploit may execute an alert box only for the purpose of a proof of concept. A real attack would strive to hijack much more vicious actions, such as, for example, money transfers, private information leakage, actions that can compromise a website and so on. Hijacking such actions can be accomplished by constructing a setup as described in the previous sections, yet this time using a slightly differently crafted URL callback parameter value. The only required modification would be to replace the "alert" string with a reference to a different method that is

bound to a more aggressive action (e.g. item purchase form submissions, publish/share buttons, admin panel actions and so on). However, in order to find this reference, an additional preliminary step is required.

Note: Typically web application functions (form submission, transfer of funds, etc) would be executed as a result of DOM events (e.g anchor/button element clicks, form submissions, registered event listeners and so on) triggered by the users or the web server itself. Whenever a user interaction event occurs, the user agent catches the event and dispatches an associated handler. The goal in the next steps would be to trick the user agent into executing a web application functionality, doing so without this user's interaction or the user's knowledge. As it was just mentioned, the first step would require assembling a reference that can later be used to hijack execution of a desired web function.

2.6.1 Predicting the Object Reference within the Target DOM

In similarity to studying an HTTP request structure and its parameters for composing CSRF[7] attacks, prior to launching the attack the attacker must perform a preliminary research over the chosen target document and compose the attack accordingly.

First of all, the attacker shall explore the target document and choose the desired web functionality. Once this has been picked, the attacker shall deduce the reference to the DOM object, which contains the method associated with this web functionality **as it would render in the target system**. Keep in mind that the DOM may vary based on the user's role, user agent version and etc.

Pinpointing the relevant object's reference in the DOM is crucial for hijacking arbitrary method execution. This holds regardless of whether the object has an explicit identifier (e.g. id attribute, script function name etc), or a reference to it can be assembled by navigating throughout the DOM tree (e.g. next/previousSibiling).

To substantiate and clarify the above, consider the following markup:

```
1  URL: http://trusted-site.com/target-document
2
3  <html>
4  <head>
5    <title>Trusted-site</title>
6  </head>
7  <body>
8      <pre id="content">
9        Bank Account:  IE92BOFI90001710027952
10       Credit card Code: 1234
11       ...
12     </pre>
13   <div id="shareBox">
```

[7] OWASP, *Cross-Site Request Forgery*, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

```
14
15      <!-- A form will appear below only for document owners -->
16
17    <form action="api/ShareDoc.php">
18      <input type="hidden" name="csrfToken" value="
            Q0jNwWaucO8axIvCCfuf0xcGWMpjiT5DlL2LSBMHd4I=">
19      <input type="hidden" name="DocID" value="87654321">
20      <input type="hidden" name="PublishDoc" value="true">
21      <input type="submit" value="Publish">
22    </form>
23    </div>
24  </body>
25  </html>
```

Listing 2.5: Example Markup of a content backup service web-page as it appears at the system of the document's owner

The markup in listing 2.5 will yield a DOM which provides many variations of executing diverse methods, for instance a form submission method associated with the functionality of "sharing a private document". Yet, as in other cases, in this example the web service would only reveal the form object in occasions where the user is indeed the owner of the document. To create an exploit that will hijack a form submission on behalf of the owners, the callback parameter value delineated in section 2.2 must include a reference to the form object.

Based on the markup in listing 2.5, here are a few variations that may be used to assemble a reference to the form object:

- **shareBox.firstElementChild**
- **content.nextElementSibling.firstElementChild**
- **document.body.lastElementChild.firstElementChild**
- *document.forms[0]*

These are just a couple of examples for using the DOM tree with an aim of assembling a reference in mind. Other valid variations would be just as efficient as these options, even though variations that will include square brackets such as the last – *document.forms[0]* would fail. The later is due to the fact that callback endpoints would usually forbid special characters as part of a callback parameter value.

2.6.2 Using the Object Reference for Method Execution

Most of the dynamic callback implementations restrict any patterns other than the alphanumeric and few other characters (usually underscore: '_' and a dot: '.' example regex may be: "[A-Za-z0-9_.]"). Thus, unless these are combined with additional bug/s and/or injections, other similar scripting attacks like XSS will be eliminated.

Despite this extensive limitation, it is truly noteworthy that a possibility to use an object reference (as given in the previous Section 2.6.1 “Predicting the Object Reference within the Target DOM”) for hijacking execution of any of its methods remains viable. Moreover, the succeeding attack often occurs without users even noticing it, since it requires no user-interaction whatsoever. Such execution remains possible as long as this method does not require arguments to evaluate. The following is a collection of such methods: clicks, form submissions, form input value tampering, defined JavaScript functions and etc (e.g. element.click(), privateForm.submit(), inputElement.stepUp/stepDown(), element.select(), element.focus(), JsDefinedFunction(), jQueryFunc() and so on.

Having said that, once the target DOM is ready, SOME exploit can, in the above scenario (Mark up in listing 2.5), fulfill the malicious intention of publishing and stealing private banking information from users. By employing the assembled "form" object reference, the exploit can hijack a form submission (sumbit method), which will in turn trigger the publishing action.

The following is an example of how to use the form object reference to execute form submission in an explicit (original) browsing context:

```
<script>
    document.body.lastElementChild.firstElementChild.submit();
</script>
```

Listing 2.6: Example form submission script

However, since the form object is a part of the target document and not the callback endpoint, hijacking arbitrary method execution such as form submission under the endpoint context would have near harmless effects. To increase the severity and risks the attack shall combine the "opener" reference created in the previous sections in order to delegate the execution context to the context of the targeted document containing this form. To accomplish such hijacking, the following shall be executed:

```
<script>
    opener.document.body.lastElementChild.firstElementChild.submit();
</script>
```

Listing 2.7: Example script to submit a form of the opener window.

2.6.3 Summary and Preliminary Conclusions

A SOME attack designed to hijack a web function, such as the "Publish" functionality that leads to stealing private banking information in the aforementioned example, will require crafting a vulnerable callback URL. The URL must be created in accordance with tree elements:

- The reference to the target document (earlier on denoted as "opener")

- The reference to the target document object which contains the desired method (e.g. "document.body.lastElementChild.firstElementChild")
- The method itself (e.g. "submit");

Thus, composing these elements into a crafted URL parameter will result in a URL similar to the following:

http://trusted-site.com/callback-endpoint?callback=
opener.document.body.lastElementChild.firstElementChild.submit

Designing the exploit to redirect the appropriate browsing context's ("WIN1" in the previous sections) document to this crafted callback URL will accomplish the described goals and pose a succeeding SOME attack threat.

With the use of the "opener" reference pointer remaining at its original allocated memory post redirection, the execution context will be designated to the target document context. The (form) object reference will be predicted and assembled by *document.body.lastElementChild.firstElementChild*. The user agent will successfully execute the desired *"submit"* method, as "Same Origin Environment" has been reconstructed since redirection. Thereby, the Same Origin Policy will be bypassed.

3 SOME Advanced Aspects

Next pages will be dedicated to an overview of some of the advanced aspects of the Same Origin Method Execution. The section emphasizes novel possibilities stemming from this attack, foregrounding the option of hijacking various and multiple web-based actions. It will also elaborate on two relevant "cross browsing context" communication techniques and discuss which of them should be seen as the most optimal choices.

3.1 More than Clicks

Same Origin Method Execution yields a variety of new possibilities. By following the steps required to exploit a vulnerable case scenario, attackers can hijack method execution of any method that does not require arguments for a successful evaluation. As opposed to other similar attacks on modern browsers, SOME provides more flexibility in terms of hijacking multiple and various web-based functionalities. The latter signifies that it does not exclusively focus on hijacking clicks.

Based on different aspects (e.g. user role, user agent version), web applications might expose or hide web functionality via producing a different DOM accordingly. The better the attackers predict the produced target document DOM structure (i.e. the form that it would appear in the target system), the greater the security risks to face. The common technique for increasing success rate implicates simulating the target system's role and learning its DOM structure (e.g. creating different test accounts in the trusted site).

Alongside the typical user interface events (e.g. mouse clicks), the attack permits hijacking **interface independent** functionality. In other words, the attack allows for a hijacking of an execution of "concealed" DOM object methods such as, for example, direct JavaScript object methods (functions), hidden elements' methods and similar.

During the BlackHat Europe 2014 Conference, the author presented a demo[1] on stealing private phone's photo albums by abusing a variation of this aspect. The exploit used SOME to hijack a JavaScript object function directly, which made the victim's user agent send the private photos to a third-party URL (external logger that saved these photos on the author's domain). The callback parameter value that led the target system to expose the private photos was crafted to execute the "PA" JavaScript function. (This function was stored as a child object property under the `pickerApp` global object). See below:

[1] YouTube, *Same Origin Method Execution Demo*, `https://www.youtube.com/watch?v=mYaNzLTb380`

```
opener.pickerApp.V.Fa.PA
```

Other hijacking possibilities may include submitting a form element (demonstrated in chapter 2 "Same Origin Method Execution"), element selection and/or focus, argument-less method execution of any object defined by different plugins (e.g. jQuery), and even actual data tampering - for instance form input values' manipulation.

In the following one can consult some examples which demonstrate the flexibility of executing what goes beyond the typical interface-dependent functions:

1. privateForm.submit()
2. element.select()
3. element.focus()
4. jQuery.jQueryFunc()
5. inputElement.stepUp()
6. inputElement.stepDown()

Note: A SOME attack allows hijacking execution of methods that does not require any arguments (as demonstrated in the examples above). Yet in a variety of vulnerable callback endpoints (especially in JSONP instances), the callback function argument may include static data which cannot be influenced externally, for example:
*controlled_callbackName(**{"data":"static_data"}**)*

Since we hijack methods that do not require arguments to evaluate successfully, such cases where the endpoint assigns fixed arguments will not clutter the attack. In essence, the modern browsers will simply ignore the arguments and seamlessly execute the given method.

3.2 Numerous Actions

In many web applications hijacking a single web function would intermittently have a meaningless effect (e.g. prompting to approve the action). One of the most outstandingly powerful aspects of SOME is the possibility to abuse a single callback endpoint for hijacking numerous actions. Consequently, it is possible to hijack a flow of actions even when they rely on each other for becoming available.

As described in the previous chapter, crafting a SOME exploit starts with designing an environment comprising windows/frames that contain documents of the same-origin. This signifies a creation of the "Same Origin Environment" (Section 2.4). Constructing

such environment requires the attacker's web-page to forge an additional browsing context. This can be done by opening a new window (using *window.open* for example) or, alternatively, achieved by creating a frame element (using a script or HTML). Because the starting web page is completely controlled by the attacker, he or she can **create more than just one new browsing context**. Consequently, an attacker can abuse all of them in his or her efforts towards hijacking multiple actions and can aim at executing a flow of actions on behalf of users.

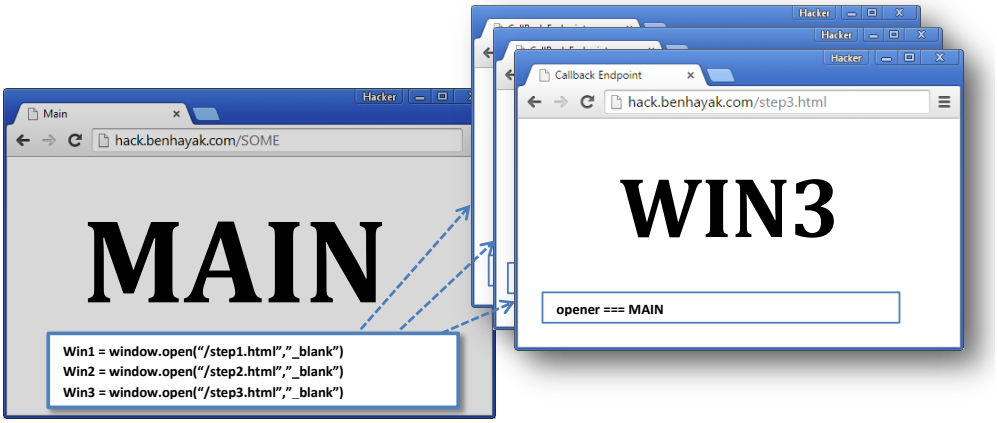


Figure 3.1: Creating multiple new browsing contexts

As described in section 2.5.1, the starting web page ("MAIN") shall be redirected to a target document hosted on a trusted site after creating new browsing contexts. At that stage, for the exploit to hijack multiple actions, each one of the new windows (e.g. win1, win2, win3 in Figure 3.1) will wait for the target document's DOM to complete loading. Ultimately, each of these new windows' documents shall be redirected to the vulnerable callback endpoint URL. However, each redirection shall be forged with a **distinctively crafted callback parameter value**. This has been described in section 2.6 (Hijacking Arbitrary Method Execution) for every action, respectively.

To clarify and specify this particular part, one shall go back to the example of using SOME to force a target system to publish a sensitive document. This time, however, the assumption is that the web application prompts for confirmation upon hijacking the publication submission. In fact, web applications would often even add a brand new form to the DOM dynamically for the user to approve the action as a prompt. In such cases, the attacker must predict this behavior and set up the attack accordingly.

Assuming that the desired objective requires a SOME attack to hijack execution of three web actions, the setup should be forged as follows:

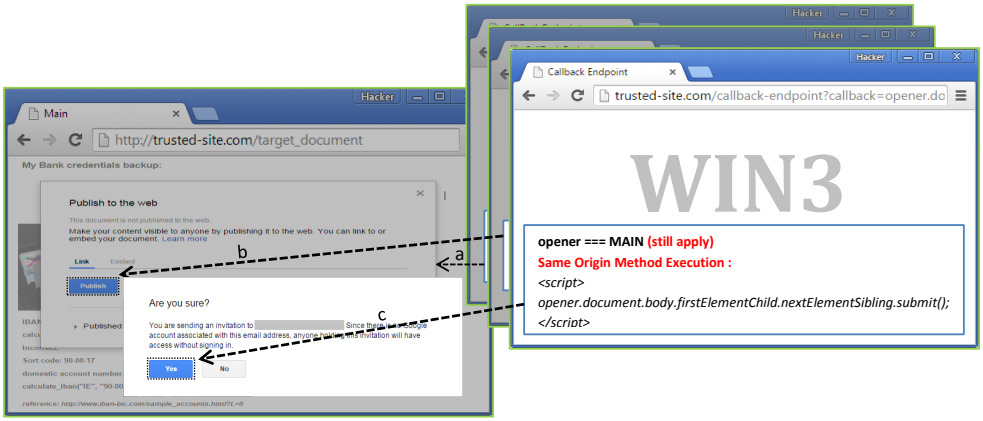


Figure 3.2: Hijacking a three-steps flow.

1. The attacker's web page needs to create three distinct browsing contexts (example demonstrated in figure 3.1).
2. The MAIN window document would be redirected to a target document URL of a trusted site.
3. Once the target document finishes loading, the created windows documents are to be redirected to a vulnerable callback URL forged according to an associated action.
   a) WIN1 would be redirected to
   http://truste-site.com/callback-endpoint?callback=opener.publishDocument
   which shall lead to the opening of the "publish" dialog.
   b) WIN2 would be redirected to
   http://truste-site.com/callback-endpoint?callback=opener.reference-to-publish-form-element.submit
   which would yield the "publish" form submission.
   c) Ultimately WIN3 would be redirected to
   http://truste-site.com/callback-endpoint?callback=opener.reference-to-confirmation-dialog-yes-button.click
   which confirms and completes the attack.

Exploiting this flow is very powerful, though can become quite complex to achieve. The actions rely on each other and, therefore, each redirection must occur only after the previous action was successfully executed. Otherwise the attack will fail. The goals can be accomplished by timing the documents (longer delay for later actions) and/or by using cross browsing context scripting (e.g. using javascript to verify the state of other windows) to redirect the documents effectively.

3.3 Cross Browsing Context Scripting

Same Origin Method Execution relies on the capacity of using scripting in modern browsers for communicating between two or more window objects. Browsers enforce the Same Origin Policy mechanism to ensure that their users can securely surf the web. Thus, when a script originating from a certain origin attempts to access dangerous properties of a document in another origin, the user agent would hinder that attempt. Keep in mind that as long as both the document from which the script originated, and the target document are at the same origin at the execution stage, cross browsing context scripting will execute successfully. SOME exploit is designed to take advantage of this behavior by designating all involved documents to a trusted site's origin prior to the execution stage.

Creating the environment (specified in section 2.4) can be done by opening several windows and/or a combination of windows and frames. Then subsequent redirecting them to the appropriate callback URLs should be performed. Both options depend on creating a new window, and therefore require bypassing the "pop up blocker" obstacle (each browser enforces its own implementation). Although since framing callback endpoints is very likely to be restricted by trusted site's servers via various frame busting techniques (e.g. X-FRAME-OPTIONS or Content Security Policy's (CSP) frame-ancestors directive), using the first proposition of employing several windows would be more reliable. Aside from this distinction, either method is absolutely suitable at this time.

User agents support cross-browsing-context communication by producing references that can be used by a new document to access its originator via window.opener property. Alternatively, in case of nested browsing contexts, this is done via window.parent property. Abusing this fact and taking advantage of these references would serve as one of the key parameters in a successful SOME attack.

4 Mitigation and Bypass

This chapter will provide a general overview of relevant current mitigation approaches aimed at protecting modern web applications and online documents from a variety of web attacks. In addition, the chapter will elaborate on several approaches employed for breaking and/or bypassing these protection mechanisms. Henceforth, most of the bypassing techniques shall be considered as exploitation optimization relevant for this attack only.

4.1 Introduction

Unless anticipated, the Same Origin Method Execution exploit will be almost completely invisible to web application mitigation techniques. This is due to the fact that the required payload only necessitates a minor and trivial set of characters (i.e. alphanumeric and a dot). The focus in the attack is substantially more pertinent to manipulating the surface rather than to the payload itself. Again, the attack solely requires alphanumeric and the dot character to hijack web application DOM functionality. Therefore, sensitization or filtering approaches, such as Cross-site Scripting (XSS) filters and Web Application Firewalls, will fail to protect against the attack. More importantly, despite some similarities shared with Cross-site Request Forgery, using CSRF tokens or checking HTTP "origin" headers will also remain impotent in terms of protecting against the attack.

4.2 Pop-up Blocker

As already mentioned in the previous chapter's section 3.3, forging a SOME exploit thus far relies upon creating a new window. In consequence, it requires bypassing the "pop-up blocker" obstacle. In order to illuminate this pivotal component, this section will discuss the pop-up blocker mechanism and provide a bypass approach.

Pop-ups were originally created for advertisement banners. The apparent original intention was to present banners without affecting the content of the currently open document. It was aimed at keeping the document "clean" from additional content, and sought to avoid information loss. Although the invention has quickly evolved, it was not a welcome transformation, since pop-ups have been largely abused by a variety of web application attacks ever since. As a response, efforts to mitigate this abnormality user agents were developed to include an additional protection mechanism, a layer commonly known as "Pop-up Blocker".

An interesting fact regarding pop-up blockers is that they are designed to block "unwanted" pop-up windows. Hence the challenge of determining which pop-up is really "unwanted" is to the the user agent to decide on (unless it is explicitly stated by the user in a different way beforehand). This automatically affects the efficiency of different pop-up blocker implementations, as the decision may vary on the grounds of the user agent brand and version code.

Pop-ups are occasionally used for legitimate purposes (e.g. OAuth pop-up dialogs), and therefore determining whether or not a pop-up is really "unwanted" can pose a challenge. Once a document attempts to open a new pop-up window, the user agent would match the URL against a white-list declared by the user. In case of a mismatch, it will block the attempt. But what about legitimate pop-up windows, such as, those which are initiated by a user click? The answer is quite simple: whenever the attempt of opening a pop-up is bound to *basic* user interaction events (e.g. mouse click, touch events), the browser will allow the attempt because of a legitimacy assumption.

4.3 Bypassing Pop-up Blocker

Pop-up blocker implementations slightly differ as far as varied user agent brands and versions are concerned. Hence bypassing divergent implementations may require slight revisions or entirely different approaches.

There are two main approaches to bypass pop-up blockers, which comprise the advance approach (i.e. requiring no user interaction at all) and a more basic approach (i.e. binding user interactions to new pop-up windows).

The most basic pop-up blocker bypass is already wide-spread, especially across the adult sites. Since user agents will allow opening pop-up windows when bound to user's click interactions, the website may simply include a false link. If it succeeds in convincing the user to click upon, it will eventually bypass the pop-up blocker and make the browser open an arbitrary window.

```
1 <a href="#" onclick='window.open("Arbitrary-URL","_new")'>Click to see
    Funny videos</a>
```

Listing 4.1: Example pop-up blocker bypass

A remaining question is why a link should be used at all in a context where there is an option to bind the functionality of opening a new pop-up window to a click event anywhere in the document.

For example:

```
1 <html>
```

```
2  <body></body>
3  <script>
4  function bypassPopups(){
5    window.open("Arbitrary-URL","_new")
6  }
7  document.body.addEventListener("click", bypassPopups);
8  </script>
9  </html>
```

Listing 4.2: Example pop-up blocker bypass by binding an arbitrary function to document body clicks.

Utilizing this bypass (or any other bypass one may find) will assist in performing the steps listed in section 2.4 with default target system configurations.

Unfortunately, the more advanced technique presented in the SOME Black Hat demonstration videos[1] has not been fixed thus far. Consequently, it cannot be covered in this paper for ethical reasons. At the same time, we remain optimistic about technically-versed readers in this area, who may already be in possession of a bypass. For others, the here described bypass constitutes quite a decent choice in a SOME exploit.

Note: Using various techniques, such as pop-under[2], can help hiding the attack and increases its success rates.

4.4 Frame Busting

The ability of framing a web-page of a trusted site has been abused by a variety of web attacks, whether the frame was used as an essential part of the attack, or by itself exposed to the UI to UI redress attacks (i.e. click-jacking). In order to mitigate this threat, several solutions were developed and implemented in different websites and user agents. Several "Frame Busting" approaches should also be mentioned. Among them one finds JavaScript frame-killers/breakers, HTTP X-FRAME-OPTIONS or CSP's frame ancestors response headers, and the like. Importantly, while their joint objective is to prevent displaying a web-page within a frame, frame-busting may have an effect on some progressive SOME exploits. Whether it is enforced or not, frame-busting may both harden and ease other bypassing stages in place.

4.5 Bypassing Frame Busting

Frame-busting has no significance in a classic SOME attack of hijacking a single web application functionality, since framing is not necessary. In fact, such case does not require a bypass at all.

[1] YouTube, *Same Origin Method Execution Demo*, `https://www.youtube.com/watch?v=mYaNzLTb380`
[2] Github, *js-popunder*, `https://github.com/tuki/js-popunder`

Although, when aiming for hijacking more than a single action, whenever the vulnerable callback endpoint (aforementioned in section 2.2) can not be framed, hijacking each action will require a new window (previously described in section 3.2). Using windows instead of frames will serve as a bypass in terms of the SOME exploit.
Note: In such case an applicable pop-up blocker bypass must be used.

In the less prevalent cases, where a vulnerable callback endpoint can be framed without restrictions, an exploit of only one pop-up window consisting of timely scheduled frames can be used to hijack multiple actions. Finding such a callback endpoint instance without frame-busting restrictions would conserve the efforts of an advanced pop-up blocker bypass, thus reiterating that an approach that does not rely on frames is more relevant and desired.

4.6 Maximum Length Limitation

Even without preventing an attack entirely, it is good practice to strive to properly secure web applications. Adding layers of security mitigation techniques operates as means of creating obstacles to attackers, and thus can often eliminate vicious attempts. In the realm of vulnerable callback endpoints, a good example of such mitigation is limiting the maximum number of characters allowed as an endpoint callback parameter value (previously mentioned in section 2.2). Setting such a limit can shrink the number of possible variations that one may use to pinpoint the DOM object reference associated with the target web function (as already mentioned in section 2.6.1). Occasionally, a maximum length limitation may even fully prevent hijacking method execution for several target documents.

In cases where the target object is located deeply in the target's DOM tree, assembling a reference might require a fairly long payload, for example:

```
1 document.body.firstElementChild.nextElementSibling.nextElementSibling.
    nextElementSibling.firstElementChild.lastElementChild.
    firstElementChild
```

Listing 4.3: 141-characters-long callback parameter payload

There is no universal technique to bypass character limitations in cases where creating references results in much too long of a payload. However, each DOM object may be reached from a variety of combinations, so a valid reference may be assembled via abbreviated approaches. To give one exemplary scenario, assembling a valid yet short reference can be acquired by "offset" navigation throughout the DOM tree. This can be done by starting from the nearest identified object reference (id object's property, default browser references, etc) aiming to reach the desired object method. Alternatively one can abuse

the existing JavaScript functions or an abbreviated version of the same target document whenever applicable (i.e. mobile versions).

4.7 Closing Remarks

Same Origin Method Execution can be exploited in several forms, therefore an exploitation shall be optimized on the basis of the targeted trusted site's restrictions. Each of the described mitigation techniques may affect the form and shape of the exploit. However, in and by themselves, they would not restrict the exploitation probability and a workaround is likely to be found.

5 Ideal Defense Approaches

The following sections will introduce an ideal defense approach which may be deployed across various layers. Conversely to the mitigation and defense mechanisms outlined so far, the risks of the attack can be entirely eliminated by the approaches presented in this section. At the same time, since most of following solutions are not completely novel, the sections will only describe the concepts without providing full guidance for deploying the protection in every environment. All the hereinafter mitigation mechanisms cannot be seen as a replacement of any of the already existing protections (especially callback parameter's character restrictions).

5.1 Introduction

Creating web applications with faster functionality and better flexibility is obviously one of the most important goals that the companies that design them aim for. Analogically, this is reflected by what consumers seek and desire to use. Fulfilling this broad yet particular need fosters inceptions of great techniques and solutions in the discussed realm. However, every once in a while, the combination of flexibility to website users and the technical solution details (cross origin data transmission for example) results in multi-faceted and highly dangerous security threats. This is exactly what happened when a technical solution favoring and granting the flexibility of controlling a dynamic callback value was introduced. Moreover, it became wide-spread as a solution for many implementations of callback endpoints, including the leading companies of Google, Microsoft, Yahoo, Wordpress and many others. Defending against these security threats would require web developers to restrain this dynamic control either by eradication and/or by matching callback values against a predefined trusted white-list.

5.2 Static Callback Values

It is already well known that in order to solve a problem, keep a high performance and spare plenty of QA testing hours, it is better to adopt an existing technique that was already tested and proven efficient, rather than to create a proprietary solution from sketch. Nevertheless, occasionally web application developers adopt every aspect of the solution without proper audit. The callback concept was created to externalize tasks and bypass Same Origin Policy, with an additional flexibility of providing a name of a function to execute for the moment when the task is completed. However, providing a parameter to control the callback value is not mandatory and sometimes, largely due to the the lack of audit, this flexible mechanism deployed by developers is more than just

unnecessary. At times, it may even expose the entire domain to security threats.

Eradicating this control by executing a fixed method in callback endpoints (when applicable) is the most efficient solution for mitigating SOME attacks.

Note: In cases where the code dynamically generates callback function names, it is strongly advised to create an additional layer that will automatically update callback endpoints with the new generated value and to remove the callback parameter.

5.3 White-listing Callbacks

As already mentioned, the callback concept is being frequently used by websites to allow cross-origin data transferring between predefined origins, doing so by executing a callback function. Needless to say, some web services expect the callback function name to change dynamically for various reasons. This occurs in cases where, for example, the company has different divisions and the callback function name is pulled from a service which is provided by another development team (e.g. compiling code with scripting compilers/minifiers or company code conventions). It may similarly happen when various functions are used to provide more than one particular service per endpoint. In these instances a white-listing option would be the next best solution.

5.3.1 Server-side White-listing

Matching a white list against parameter values in the server-side code is pretty basic and likely to be deployed in websites on various web features that include multiple choices. For that reason, as well as in consideration of the fact that each website may use different server-side scripting languages, the code sample in listing 5.1 would be just an example of matching a white-list. Thus, it is not intended to serve as the best approach delineation:

```
1  <?php
2  /*[...session verification...]*/
3  if($_GET["callback"] === "cbTask1") {
4    //execute callback code ;
5  }
6  elseif ($_GET["callback"] === "obj.cbTask2") {
7    //execute callback code ;
8  }
9  else {
10   exit();
11 }
12 ?>
```

Listing 5.1: basic PHP white-list approach

Note that the example in listing 5.1 covers parameter values provided by HTTP GET requests, although the "GET" string may be replaced with "POST" for values provided by HTTP POST requests.

5.3.2 Client-side White-listing – Registering Callbacks

A server-side white-list solution may not be sufficient for two main reasons. Firstly, due to the fact that there are many diverse server-side scripting languages paired with the use of proprietary languages created in companies internally. Secondly and most importantly, the vulnerability can also be found residing in the fully client-side systems, such as Flash applets. A generic server side white-list solution for Same Origin Method Execution might turn out to be too complex or ineffective to deploy. However, an ideal generic solution can be achieved through deployment of a client-side code that allows registering a list of callback function names via client-side scripting. The latter proposal would guarantee a decent white-list mechanism whilst keep the flexibility of supporting both the dynamically generated callback names and client-side plugin systems (e.g. Flash applets).

The following is a JavaScript solution for registering callback function names under a generic object that will hold a list of registered records only:

```
URL: http://example.com/callback-endpoint?callback=alertCallback

<script type="text/javascript">
function simpleAlert(){
  alert('example callback');
}
__SOME__ = {};
/**
 * Register callback functions
 * @param {String} cb_name    /* callback name to register *
 * @param {Object} ctx      /* object scope of the function  *
 * @param {Function} target_function /* function to associate with the
     callback name *
 */
__SOME__.register = function (cb_name,ctx,target_function) {
  if (typeof(target_function) == "function") {
    __SOME__[cb_name] = target_function.bind(ctx);
  }
  else {
    console.log("Error while providing parameter values ")
  }
}
__SOME__.init = function() {
  __SOME__.register('alertCallback',this,simpleAlert)
}();

/**
 * SERVER-SIDE CODE
 */
<?php
/* [...HTTP callback parameter sensitization and restriction...] */
  echo "__SOME__['" . $sanitized_cb_value . "']();";
?>
</script>
```

Listing 5.2: basic example of registering callbacks in JavaScript

Following a deployment of the client-side solution suggested in listing 5.2, callback endpoint developers can register new callbacks dynamically using the *__SOME__.register* function and reference them using *__SOME__['parameter_value']*. This technique will match any value provided via a callback parameter against records saved in a global (*__SOME__*) object. It will also make sure that only the registered callbacks can be executed.

Note that the code in listing 5.2 was minimized to provide a clean example of the concept for the purpose of this paper. In consequence, the code may include object prototype chain's security-related issues. Keep in mind that following the completion of the thesis, more work will be done on the ongoing solution-focused project by the author.

Note: More details/updates about the ongoing project will be revealed under the following github project: https://github.com/BenHayak/SOMEGuard

5.4 Cross-Window Messaging

User agents enforce Same Origin Policy rules and thereby restrict cross-origin document access. To bypass this restriction and communicate between cross-origin documents, web services may use the callback concept. At the same time callback method execution is by no means the only solution. A JavaScript API called postMessage[1] can be utilized for transferring messages between cross-origin documents. Likewise, it can be employed to transfer data and/or notify a service upon an event of a different window's document. Although, such deployment will require additional efforts of various composite verification and security-oriented checks.

Note: The Cross-Window Messaging solution is only relevant for vulnerable callback endpoint documents. Therefore, it is not an applicable resolution for vulnerable plugin systems, such as Adobe Flash applets.

5.5 Conclusions

Same Origin Method Execution is an attack that takes advantage of the control granted by an external callback parameter. Deploying any of the aforementioned mitigation mechanisms, on either server or client-side, will restrain the external control and consequently eliminate the attack. It will further eradicate other callback-related attacks such as "Rosetta Flash" [2] (by Michele Spagnuolo) and possible future attacks.

[1] MDN, *Window.postMessage*, `https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`

[2] miki.it, *Rosetta Flash*, `https://miki.it/blog/2014/7/8/abusing-jsonp-with-rosetta-flash/`

6 Appendix

6.1 Acknowledgements

This work has been supported by Trustwave Inc. My gratitude also goes out to Nir Goldshlager, Oren Hafif, Daniel Chechik, Shai Rod, Aviad Golan and Arseny Levin for their valuable insights and comments pertaining to this research.

Several aspects of Same Origin Method Execution have been discovered by Google's Aleksandr Dobkin and LinkedIn's Roman Shafigullin. They both certainly deserve credit for their related work.

Thanks to Google's Eduardo Vela and the entire Google security team for an outstanding bug bounty program. It allowed me to demonstrate this attack against a significant web service, namely Google Plus. Many thanks goes to Paula Pustułka for proof reading, advice and support.

Finally, thanks to Dr. Mario Heiderich for advising and supporting this thesis - many thanks for dedicating valuable time and providing guidance on many aspects of the project.

6.2 Related Work

The following links point to great research works related to the attack covered by this thesis:

- Reverse Clickjacking by Aleksandr Dobkin
  - https://plus.google.com/+AleksandrDobkin-Google/posts/JMwA7Y3RYzV
- Same Origin Method Execution in Microsoft by Jakub Zoczek
  - http://zoczus.blogspot.co.il/2015/01/yammercom-same-origin-method-execution.html
- Same Origin Method Execution in Google Plus by the author
  - http://www.benhayak.com/2015/05/stealing-private-photo-albums-from-Google.html
- BlackHat Presentation Slides
  - https://www.slideshare.net/BenHayak/blackhat-eu-same-origin-method-execution

Listings

List of Figures

Curriculum Vitae of Ben Hayak

Contact Data

E-Mail ben.hayak@gmail.com

Military Service

2007-2010 Data Computer Network Communications technician and Network Security (Air Force), "CCNA" and "CCNP Route" qualifications.

Professional Experience

2010–2013 Information Security Consultant and penetration tester, Avnet Information Security.

2013–2015 Security Researcher, Trustwave.

since Apr. 2015 Senior Product Security Engineer, Salesforce.

Publications and Awards

1. Google Security Top 0x0A
   - http://www.google.com/about/appsecurity/hall-of-fame/
2. White-hat cyber bug bounty nets cash
   - http://nypost.com/2012/08/26/white-hat-cyberbug-bounty-nets-cash/ (August 2012)
3. Google, Facebook , Twitter, Microsoft, eBay, Paypal, Adobe and DropBox white-hat acknowledgments.

Conference Talks

1. Bitcoin Transaction Malleability Theory In Practice – How transaction malleability works in practice, the technique presumably used for stealing MtGox bitcoins, Ben Hayak, Daniel Chechik, Black Hat 2014, Nevada, USA.
2. Same Origin Method Execution – Exploiting a Callback for Same Origin Policy Bypass, Ben Hayak , Black Hat 2014, Amsterdam, Europe.

Projects and Work

- Penetration-Testing for various companies
- Various memory corruption zero days research and analysis
  - https://www.trustwave.com/Resources/SpiderLabs-Blog/The-Kernel-is-calling-a-zero%28day%29-pointer-%E2%80%93-CVE-2013-5065-%E2%80%93-Ring-Ring/

  - https://www.trustwave.com/Resources/SpiderLabs-Blog/Deep-Analysis-of-CVE-2014-0502-%E2%80%93-A-Double-Free-Story/

- Further references can be requested

Technical Document submitted to

BlackHat UBM conference
Nov 2014

Date of publication, May 2015