National Conference on Recent Trends in Computer Science and Technology (NCRTCST)-2013

Improving Modern Web Application Defenses Using HTML5

Achin Kulshrestha
Bangalore, India
Email: achinkul@gmail.com

Abstract – **HTML5 has brought in several technological changes to the erstwhile HTML realm and even though there are certain security implications due to the introduction of various new APIs, considerable amount of effort has been put to corroborate secure Web application development methodology. The purpose of this paper is to elucidate different HTML5 features that can be used for the benefit of improving the state of security germane to Modern Web Applications. This research also focuses at the grassroots the intricacies of popular HTML5 API's and their secure usage.**

Keywords – **CORS, HTML5, Injection, Security, Web.**

I. Introduction

HTML5 has carved a new pathway for the next generation web applications. Today, there is huge number of changes happening to empower the web and take it to the next level. HTML5 is improving browser capabilities leaps and bounds and with technologies like XMLHttpRequest (XHR), Document Object model (DOM), Cross Origin Resource Sharing (CORS) and HTML rendering enhanced drastically than their erstwhile versions several advanced technologies such as Webstorage, WebRTC, Websocket, to name a few, are creating tumult all over the web industry. With the advent of HTML5, new security issues have also begun to come up and also novel attack points for attackers with malicious intent. However, during the making of the HTML5 specification security considerations were made from the very beginning and that has led to some major improvements that are really important in making a modern web application secure to the maximum limit.

II. Script Injection

A. Demystifying XSS

Cross Site Scripting [1] or XSS is one of the most common web attacks that attackers use to get into applications today. The same origin policy for javascript postulates that if there is XSS vulnerability anywhere on the domain, everything on the domain is vulnerable. Even one XSS weakness give a huge advantage to attackers to run amok in the browser, impersonate users and do anything which a javascript code can do. Most of the attacks involve two entities – the attacker, and the web app, or the attacker and victim client that is the web browser, the XSS attack involves three entities – the attacker, a browser and the web app. XSS results from the intermixing of the server code and user input. If user input is not filtered and sanitized correctly, it could contain code that executes along with server code in the victim's browser.

XSS attacks can be very severe. Typical XSS payloads include Cookie theft, password logging, Worm distribution, Phishing and the ability to gain entry onto an internal network environment [2].

There always have been security measures that can be taken to protect the application from XSS attacks such as restricting untrusted javascript, not allowing external websites from requesting internal resources. However, with HTML5 a new feature known as Content Security Policy was introduced which significantly reduces the XSS attack consequences.

B. HTML5 Content Security Policy for Fighting XSS

Content Security Policy (CSP) [3] is a technique that allows application developers to whitelist the sources from which applications can load resources. The approach behind this feature is the use of specific directives which help in locking down the application by restricting various web resources for the application. The primary objective behind Content Security Policy is to mitigate content injection vulnerabilities XSS vulnerabilities tend to be the most prevalent and harmful form of these vulnerabilities and in 2013 is ranked the #3 web application security risk by the Open Web Application Security Project (OWASP). CSP allows web app to supply a Content-Security-Policy HTTP header that specifies whitelisted sources from which scripts can be loaded/connections can be made. When the web application is trying to load a script, the browser compares the source of the script against the Content Security Policy directive before loading it. CSP is not just limited to script origins; it also has the capability to control the sources of images, stylesheets, XHR, and Websocket connection before allowing them to go through load. Indeed, CSP is a second line of defense and should be only be used as an added security mechanism for the web application. Application developers should still emphasize on proper filtering and output encoding of data to mitigate class wide XSS attacks.

C. Best Practices for CSP implementation

1) A default-src directive should always be specified

It is always better to start with full lockdown and then start allowing resources that are absolutely needed. To do that set the default-src directive to 'self' or 'none' and then start adding other directives like script-src, media –src etc. as needed. This would help in making the web application secure at the grassroots with additional directives serving as exceptions added in order to maintain functionality. This also help in achieving flexibility because if any new source location has to be added/removed, it can be done very easily.

2) Avoid the Unsafe-inline Directive

The biggest source of script injection in the DOM is inline scripts. Since CSP only allows the server to completely control the sources from which the app expects to get resources from. With the presence of the unsafe-inline directive CSP would ignore any inline scripts and thus it has no way to control the injection of inline scripts. Ultimately, it is the responsibility of the developers to remove inline scripts from their code as they are significant source of XSS attacks.

3) Avoid the unsafe-eval directive

If the input to a Javascript function eval() is not sanitized, it is an unwise decision to make as any input will be executed in the current javascript context. CSP prevents the use of specific Javascript functions such as eval() for added security since it can be used to execute user input as code. Adding the unsafe-eval directive allows the eval function to be used by the application and hence, bypasses one of the most critical aspects of Content Security Policy.

4) Wild Card (*) should not be used as the Default-src policy

CSP allows wildcards to be matched through the use of '*'. A Content Security Policy containing just '*'as the default policy is same as not having CSP header at all. In this way it will not be able to protect against any injection risk.

*5) Do not specify a Wildcard when referencing Top level domains such as *.org*

Wrong usage of a wildcard in the CSP header can have very bad effects for the application as the server could allow access to more resources than were ever intended. A default-src *.org directive would allow anyone with a .org domain to include their content in the victim application.

III. HTML5 Mashup Security

With HTML 5 the current web development process has moved leaps and bounds from the server side generated content to client side generated. Application are utilizing frameworks like HTML5 Boilerplate, twitter Bootstrap that use massive amounts of JavaScript and CSS for generating beautiful looking and responsive user experiences. Ultimately, this leads to the situation where developers want to include or request third party resources. Before HTML5, the Same-Origin-Policy of the browser did not at all allow resources to be loaded cross domain. This policy strictly specifies that the client side Javascript code could only request resources from the origin it being executed from. To be more lucid, this logic is predicated on the notion that a script from abc.com can't load any resource from xyz.com using XMLHTTP Request. Or in other words Ajax. As a consequence, to support this required functionality, the W3 consortium extended the specification that through usage of a special set of headers a resource can be retried cross domain. In contemporary jargon, this is known as Cross-Origin Resource Sharing (CORS) [4]. The headers which pertain to CORS are the Access-Control-Allow-* header family.

The most critical header for CORS is the Access-Control-Allow-Origin header that specifies from what all origins access to the requested resource is allowed. By default, if a script is executed from another origin, the browser typically raises an exception and drops the response received from the server side. Since in both the scenarios, the request is sent to the server so server side code is involved in both the cases.

To prevent this overhead, the specification includes an specific flow so that the if CORS is not supported, the connection is dropped much before that. To achieve this, the browser sends an OPTIONS request known as the Preflight request to the server that holds the resource. If the browser finds from the preflight response that the actual request to be sent would conflict with the CORS policy, an exception is raised and the original request is never sent to the server. On top of this, the OPTIONS response could also include a second important header for CORS which is the "Access-Control-Allow-Credentials". This header allows the browser to send authentication data in form of HTTP based authentication or cookies.

Since the crux of this HTML5 technology lies on the Access-Allow-Control-Origin header, it is necessary that the developers take care that no wildcard is used to define the directive for this header

A. Communicating Between Iframes

There are cases when two different origins have to be loaded in the same DOM and some communication between them is required. HTML5's Web messaging [5] is a way for documents in different browsing contexts to share data without the DOM being exposed to malicious attacks like cross-origin scripting. Unlike other type of cross origin communication available like XHR, web messaging doesn't expose the document object model directly.

As per Browser's Same-origin-policy If we want to some data to an advertisement embedded in an iframe, served by a third-party origin. If our parent DOM wants to access a variable within the iframe or vice-versa, a security exception will be thrown. The message event functionality of Web Messaging allows different origins to communicate securely if properly used. Cross-document messaging is often referred to by its syntax as window.postMessage().

B. Secure Implementation of Cross Domain messaging

To send a cross-document message using web messaging we need to create a two different browsing context as in a new window or an iframe. The message can then be sent using the postMessage() method. The Postmessage API requires two arguments, one is the message to be sent and other is the Origin of the target. When this message is sent, the browser automatically sets the postmessage object with the value of the source origin. So the receiver should have code to make sure that the request is coming from a legitimate origin. Similarly, for the response the source origin can be verified.

IV. HTML5 Sandbox

As explained in modules above developing a rich application on today's web makes it necessary to embed components from third party sources. Abstaining from use of mashups isn't really an option for Modern Web application. However, by doing so, the risk of a security breach significantly increases. Each component that is embedded is a potential attack point for those with malicious intent. HTML5 provides a feature called Iframe sandboxing [6] that allows running third party component with least privilege.

A. Sandboxing for allowing Least Privilege model

Essentially, the aim is to embed the required third party component and give it only the minimum level of capability necessary to do its task. If the component doesn't need to use javascripts at all, taking away the functionality to run JS in the context wouldn't hurt. To be more specific, if we don't require flash to be run, it would be better if we can disallow running of a plugin in the component. The principle of least privilege is the most secure way achieving the desired goal but at the same time making sure that critical components are secure to the best possible extent. This helps in making sure that no blind trust has to be made on the embedded content and it would only be allowed the functionality which it absolutely needs.

Iframe, though a relatively old technology were the stepping stones towards a possible solution for providing least privilege. If some third party content is loaded in an iframe, it provides a measure of separating the original application and the untrusted content. The content which is present in the iframe will not have access to the parent's DOM data. However, this separation isn't truly foolproof. The iframe which is running untrusted content still can do malicious things like popups which can lead to phishing attacks etc.

With HTML5 sandbox attribute the iframe element gives exactly what is required to tighten the iframe's capability allowing only the subset of capabilities necessary to do whatever work needs to be done.

Sandboxing works on the basis of a whitelist. So similar to CSP we begin by removing all permissions possible, and then allowing individual capabilities to the sandbox's configuration.

B. Understanding Sandboxing Attributes for Secure implementation

For an iframe with an empty sandbox attribute
<iframe sandbox src=""> </iframe>
the document inside the iframe is fully sandboxed and tightened with the following restrictions.

- No JS execution.
- Frame is loaded into a unique origin, effectively no origin.
- The iframe cannot create new documents using window.open = _blank
- Forms cannot be submitted.
- Plugins can't be loaded.
- The framed document can only navigate itself, not its top-level parent.
- Features like autofocus are blocked.
- However, putting all these restrictions mean that the iframe can't possibly do anything and for that purpose iframe sandboxing provides granular control.
- allow-forms allows form submission.
- allow-popups allows popups.
- allow-same-origin allows the document to maintain its origin;
- allow-scripts allows JavaScript execution
- allow-top-navigation allows the document to navigate to the top-level window.

One catch here is that the sandboxing flags applied to a frame also imply the same flags would be applied to frames created in the sandbox.

V. Conclusion

On our path to the zenith of web usability, we are seeing thin lines of demarcation between flexibility, security and privacy. The HTML5 standard has introduced some amazing concepts but at the same time the attack surface for application has been increased as well. It is true that HTML5 is making progress in retrofitting security features in the specification but it will require a lot of work to be make it robust. The web is the largest vector for distribution of worms, viruses and other malicious activities. As web technology steps into the domain of performing unimaginable activities which only desktop application were thought to be capable of doing, the security risks are going to increase and simultaneously the defenses against them.

References

[1] Amit Klein, "Cross Site Scripting Explained," URL: http:// crypto.stanford.edu/cs155/papers/CSS.pdf, pp.1-3,June 2002.
[2] Doug DePerry, "HTML5 Security, the Modern Web Perspective", URL: https://www.isecpartners.com/media/18610/ html5modernwebbrowserperspectivefinal.pdf,pp.3-12.Dec. 2012
[3] Francois Marier, "Defeating cross-site scripting with Content Security Policy," URL: http://wellington.pm.org/archive/ 201202/francois-marier_csp.pdf, 2012 .
[4] Timo Schmid, "Security Imapacts of HTML5 CORS," URL: http://www.insinuator.net/2013/08/some-security-impacts-of-html5-cors-or-how-to-use-a-browser-as-a-proxy/, 2013.
[5] Tiffany Brown, Opera Developer Article, "An Introduction to HTML5 web messaging", URL :http://dev.opera.com/articles/ view/window-postmessage-messagechannel/, 2012
[6] Mike West, "Play safe in Sandboxed Iframes," URL: http://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/, 2013.