Automating DOM-based Cross Site Scripting Protections on Chromium and Chromium-based browsers

(Final Technical Report)

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Dale Wilson

Spring, 2020

Department of Computer Science

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: Dale Wilson

Approved: _____ Date _____

Richard Jacques, Department of Engineering and Society

Approved: Yuan Tian Date 04/25/2020

Yuan Tian, Department of Computer Science

Introduction

Security thrives as a field built to prevent, mitigate, and predict the occurence of danger. In an age dominated by digital infrastructures that manage how we socialize, travel, and eat, old and nascent technologies must grow to secure the modern digital landscape. A collective push for digitizing commercial and nonprofit organizations stresses the need for comprehensive security measures across an ever-increasing number of distributed web applications.

The technical report focuses on a DOM-based Cross Site Scripting (DOMXSS for short) defense for Chromium, the open source project upon which Google Chrome is built. XSS attacks occur when benign websites are distributed with unsanitized user controlled inputs, these can execute as malicious code on an unwitting user's internet browser. Creating an adequate defense for DOMXSS is uniquely challenging because attacker controlled malicious inputs may never leave the client (the internet browser); this suggests an effective protection must occur in a browser engine update and input filtration at runtime.

Technical Topic

XSS attacks have been one of the most prevalent threats to the modern web over the past decade. Web development has progressed in a direction where heavy javascript is executed on the browser, exposing the browser to more Document Object Model-XSS (DOM-XSS) attacks that could be undetectable by servers distributing web applications. This shift in design paradigm isn't entirely bad, it provides end users with more robust, real-time interactive web applications. In affording developers more freedom to update the client-side state of a web

application, the Document Object Model (DOM), the modern web has grown into what the world recognizes it for today.

Detection of these DOM-XSS attacks leverages taint tracking to identify whether data from attack-controlled sources can reach sensitive sink functions (Melicher W., Das A., Sharif M., Bauer L., Jia L. 2018)). These sensitive sink functions directly modify the DOM on the browser. This attack surface can be abused in a variety of ways but the vectors of greatest interest are those by which an attacker can execute code via URL-based sources. Methods used by CMU's DOM-XSS research projects have found 83% more vulnerabilities than that of previous studies, indicating that DOM-XSS attacks can abuse an increasing attack surface (Melicher W., Das A., Sharif M., Bauer L., Jia L. 2018).

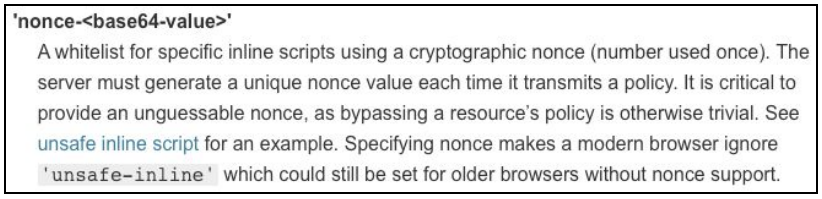
Recent research proposes client side vulnerabilities should be split into two subclasses: DOM-based and persistent client local storage vulnerabilities (Steffens, M., Rossow, C., Johns, M., Stock, B. 2019). This exemplifies the weaknesses that need a fix; sensitive functions that have the capacity to manipulate the web application's state from the client side, the DOM. In a modern web, these functions must continue to manipulate the DOM while preventing user controlled input from potentially escaping a function's context and executing as malicious code. It makes sense to approach the defense by altering a browser's script creation and processing behavior in addition to modifying how these functions operate at runtime; a runtime defense allows for flexibility via an opt-in system where as an exclusively browser-based approach is harder to opt-out of in cases of incompatibility.

The UVA research project aims to contribute DOMinatriXSS, a defense against DOM-XSS (Tian Y. 2015). There are two components to the project: DOMinatriXSS, an

externally loaded JavaScript (JS) library maximizing the defense's adoptability, and DOMinatriXSStatic, the Chromium project contribution both enabling the 'disable-dynamic' CSP directive and updating the browser engine.

I finished development of the DOMinatriXSS JS library first, as its implementation operates nearly independent of the browser implementation. I broke the development process of the DOMinatriXSS JavaScript library into three phases, (1) modifying/injecting a meta tag to enforce generated script nonce, (2) the inline event handler conversion of scripts to event listeners, and (3) adding the generated script nonce to imported, dynamic JavaScript libraries via document.createElement. These three phases parallel the three core components of the library. The library is meant to automate protections and a secure coding style that a web developer should have already adopted, it ensures security from injections via document.createElement() and inline event handlers written into markup. The rewrite of insecure inline code as well as nonce propagation maximizes compatibility with progressive web application frameworks like Angular and React or other developer-friendly tools that leverage DOM manipulating functions. Since the 2013 addition of nonces as a source to the 'script-src' CSP directive, both unsafe-inline and a script nonce should be applied to securely rewrite inline event handlers externally, if only of these two sources is defined then the rewrite will not occur properly (Matatall, N 2013).

Image 1: Screenshot from Mozilla's Web Documents detailing the specific policy interaction between 'unsafe-inline' and script nonces



The current implementation of the JS library is a bit different from the 2014 version. In the current version, the content of the meta tag applied must also include 'unsafe-eval' in order to add inline event handlers using the .addEventListener() function (as seen in image 2). The generated script nonce is not added to each event listener as they are allowed to execute if rewritten externally as accomplished by the library. The library then strikes 'unsafe-eval' from the page's meta tag content to remove the eval() function from the attack surface.

Image 2: Screenshot of inline event handler conversion attempt when 'unsafe-inline' is not included as part of allowed script sources in content security policy

The development process for the DOMinatriXSStatic implementation was broken into three phases, (1) updating the function that creates document fragments, (2) updating the script runner object that runs all script elements on a webpage, and (3) adding the new

‘disable-dynamic’ CSP directive to the Content Security Policy object in Chromium’s browser engine. Upon document fragment creation, a flag (private variable) is set for the document fragment indicating whether the ‘disable-dynamic’ CSP directive was present. In the ParseHTML function, the ParseDocumentFragment() function is passed the ParserContentPolicy parameter ‘kDisallowScriptingAndPluginContent’ if the ‘disable-dynamic’ flag was set, otherwise the function is passed the original parser_content_policy parameter. The html_parser_script_runner object is updated to prevent scripts of nesting level higher than 0 from executing if ‘disable-dynamic’ is specified, there is already an object called html_parser_reentry_permit which keeps track of a script’s nesting level, so I added a function that will return the object’s script nesting level. If the ‘disable-dynamic’ CSP directive is specified, the html_parse_script_runner object returns early from its ProcessScriptElement() function to prevent the injection of a potentially malicious script. It seems as if the blink browser engine is not so drastically different from the original webkit browser implementation such that the strategies for securing .innerHTML, .outerHTML (via document fragments) and document.write() (via script nesting level) would work.

For testing the implementations of DOMinatriXSS and DOMinatriXSS, I’ve begun to use the community edition of the Burp Suite platform The automated web vulnerability scanner is priced at $400 USD a year, so further testing how the current implementation fares when used with modern web applications as opposed to the web applications built in 2014 may be required. Due to a technology transfer process, the company that built DOMinator (the web scanner used by the 2014 investigation for testing) unfortunately does not have license keys for their testing platform available (once only £60 GBP).

Future Work

The goal in creating the JavaScript library is to automate some of the work necessary to prevent DOMXSS, enforcing script nonces on served web pages. By creating a lightweight library that applies script nonce protections and rewrites static inline event handlers (an insecure but frequent way of writing frontend software) DOMXSS can be prevented without the developer having to significantly change their development style. Open sourcing the JS library in the near future, distributing it via CDN, or publishing NPM/PIP/GEM packages that will embed the library in served web pages could afford developers an efficient-to-deploy defense for DOMXSS.

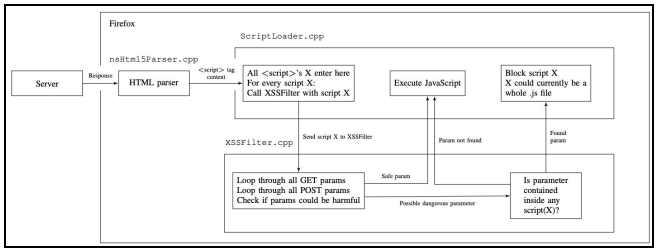
The portion of the defense that updates the blink browser engine could eventually be contributed to the Chromium project, the open sourced project upon which Google's Chrome and Microsoft's edge are built. In order to do so, it could be necessary to further test the compatibility of the defense with more modern web apps, as the previous testing was conducted on web apps developed five or more years ago. Before contributing the updated code must also meet the stylistic coding standards of the open source project.

The current project provides an approach to updating the blink browser, but Chromium's old, previously shared browser engine webkit retains similarity to the blink engine. This could lead to a similar implementation of the current DOMXSS protection within the webkit engine, the underlying project upon which Apple's Safari is built. Because the current project's

inception occured when Chromium still used the webkit engine, a similar implementation for the current webkit engine is certainly within reasonable scope for future work.

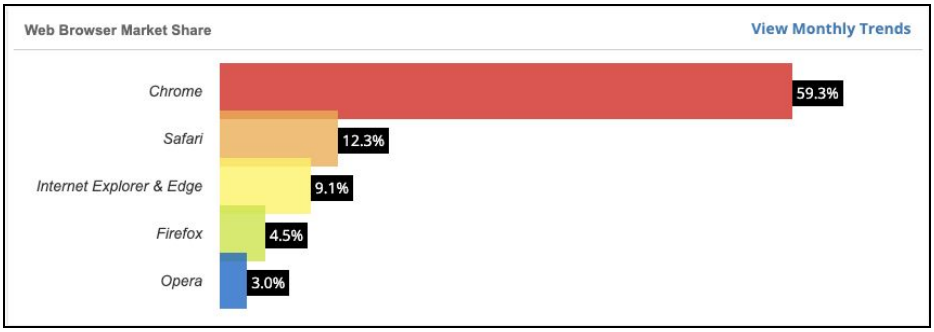
While the blink and webkit browser engines are quite similar, Mozilla's Firefox is built upon the gecko browser engine. Recently proposed Client-Side filters for DOMXSS attacks on the Firefox browser use regexes and string matching (Vikne, A. and Ellingsen, P. 2018). Vikne and Ellingsen's filter takes inspiration from Chrome's XSS Auditor but differs as the proposed Firefox defense executes matching only on scripts to be processed by its internal script handler, while the Chrome auditor executes matching on every single DOM tree node. Both of these defenses rely on matching which has the potential for large overhead, slowing down page loads due to unnecessary extra work accomplished by the browser, as shown in image 3 displaying the proposed update to the Firefox ScriptLoader object (Vikne, A. and Ellingsen, P.). I think this is promising in showing how the proposed defense for Chromium is not only lightweight, but also effective in attacking the problem of injecting scripts before they even reach a respective browser's script loading object.

Image 3: A diagram of the proposed matching defense for the Firefox browser engine (Vikne, A. and Ellingsen, P. 2018)



With a similar implementation of DOMinatriXSS across the blink, webkit, and gecko browser engines, it could be possible to achieve coverage for the proposed DOMXSS defense on up to 85% of all web browser activity.

Image 4: A bar chart displaying internet browser market share for the most popular browser (https://www.w3counter.com/globalstats.php)



Conclusion

The technical aspect of my project focuses on securing the digital experience of users on platforms like Google Chrome from attacks undetectable by servers. The successful integration of the technical research in open-source projects like Chromium could directly impact millions of users by limiting the current vectors for DOM-XSS attacks.

References

Burp Suite - Cybersecurity Software from PortSwigger. (n.d.). Retrieved from https://portswigger.net/burp

CSP: script-src. (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src

Matatall, N. (n.d.). Re: CSP 1.1: Nonce-source and unsafe-inline. Retrieved from https://lists.w3.org/Archives/Public/public-webappsec/2013Jul/0027.html

Steffens, M., Rossow, C., Johns, M., & Stock, B. (2019). Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. *Proceedings 2019 Network and Distributed System Security Symposium*.

Tian, Y. (2015). DOMinatriXSS: Automated DOM-Based Cross-Site Scripting Protection. *Network and Distributed Systems Security (NDSS) Symposium 2015*.

Vikne, A. and Ellingsen, P. (2018) Client-Side XSS Filtering in Firefox. *SOFTENG 2018 : The Fourth International Conference on Advances and Trends in Software Engineering.*

William, M., Das, A., Sharif, M., Bauer, L., & Jia, L. (2018). Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. *Network and Distributed Systems Security (NDSS) Symposium 2018*.

W3Counter. (n.d.). Retrieved from https://www.w3counter.com/globalstats.php