Cross Site Scripting in Front End Web Development:

How modern front-end web development opens up XSS holes

Teddy Cleveland

I. Abstract

Over the past few years, web development has seen a decisive shift from applications developed under single, comprehensive frameworks such as Rails to a more distributed approach: one in which the front and back-ends are grown as distinct entities exchanging application data. As a result, many back-end applications that used to handle templating and DOM manipulation now rely on front-end frameworks such as Backbone.js, React.js, and Ractive.js to manage web-page rendering and more importantly, dynamic web-page manipulation. In addition, many of these emerging frameworks seek to be "opinionless", resulting in libraries that serve as simple organizational tools and leaving an increasing amount of responsibility to front-end developers to provide core security features. Because many front-end frameworks rely heavily on direct DOM manipulation using Javascript (often jQuery) without built-in sanitization, and because much of the rendering code is often written by front-end web engineers without backgrounds in security, this paradigm shift towards front-end frameworks opens up a plethora of vulnerabilities to cross site scripting (XSS) that must be addressed. As front-end Javascript frameworks grow less opinionated and rigid, more onus falls upon the developer to address the XSS issues inherent in direct DOM manipulation. In this paper, I will discuss XSS security flaws as the pertain to front-end Javascript development. I will also provide a security layer, which I call Input Sentinel, that allows developers to handle many basic XSS attacks by monitoring HTML forms, even when they are added to the page dynamically.

II. Introduction

So what is Cross Site Scripting (XSS)? XSS is an injection technique whereby attackers take advantage of an application's ability to send information to users. Instead of posting something like a Facebook status, the attacker posts malicious Javascript code, that when executed in another user's browser can do serious harm. This issue arises primarily when an application does not sanitize data and can even happen persistently if the malicious code is stored in the application's database.[1]

A proper inspection of modern front-end development necessitates a look back at some of the tools that are being abandoned in favor of more modern Javascript frameworks. Because pure js

[1] "Cross-site Scripting (XSS) - OWASP." 2008. 15 Dec. 2015 <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>

frameworks provide a more dynamic page rendering process, they can be much more versatile and far faster than traditional mold-injection styles of templating. Comprehensive "full-stack" frameworks like Ruby on Rails and Django each provide sanitization features that prevent most harmful data from being rendered into the DOM as HTML. Rails provides a *sanitize*[2] method for cleaning input and escapes specific characters that are not whitelisted. Django, too provides similar sanitization features to prevent XSS when rendering templates into the DOM.[3] Because the communities surrounding these frameworks are fairly large, the source code is well-maintained, well-documented, and implements several layers of basic security. Despite the comfort of full-stack frameworks however, we are seeing a decisive shift towards a separation of front and back end code into a diverse array of frameworks--with new Javascript frameworks seeming to pop up every year (Angular, React, Ember, Backbone, Ractive, and many more)[4]. The shift towards Javascript as an application language stretches security concerns even further across the entire stack, now requiring that front-end developers address these issues separately from those addressed on the back-end.

One of the principal attractions of many prolific Javascript frameworks is that they provide the ability to build dynamic web pages--that is to say web pages that change on the fly as users interact with them, with only small sections of the page being updated in between full page renders. Some frameworks even go so far as to support single-page apps, where the page is only really rendered once, and future page changes are implemented as direct DOM updates. While live changes to the DOM provide great opportunity for creative design, they open up a host of vulnerabilities to XSS attacks (among many other injection assaults). Depending on the application, users can often update the page (or even the database) with malicious code even when a framework like Rails may have prevented it. The true difficulty in addressing this situation is that the insecure nature of dynamic DOM manipulation can be considered a feature in some situations: developers may want to load scripts into the page dynamically.

Front-end frameworks often don't enforce security past their own functionality, leaving organizational decisions, dependency loading, and consequently security up to front-end developers. The choice to use a Javascript framework or roll your own front-end application means accepting the responsibility to address the related security concerns.

---

[2] "ActionView::Helpers::SanitizeHelper - Ruby on Rails API." 2007. 15 Dec. 2015 <http://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html>

[3] "Security in Django | Django documentation | Django." 2014. 15 Dec. 2015 <https://docs.djangoproject.com/en/stable/topics/security/>

[4] "Front-end JavaScript frameworks · GitHub." 2014. 15 Dec. 2015 <https://github.com/showcases/front-end-javascript-frameworks>

III. But Why does it Matter? (To the Community)

Of course, the discussion of security as a front-end issue begs the question, "why does it matter?" In a post on Web Design Weekly entitled "Front End Security is a thing, and you should be concerned about it", Tim Evko discusses an XSS vulnerability in Tweetdeck that resulted in the site's paralyzation for a short time, afterwards asserting "In case it hasn't been made clear already, front-end security is an important issue."[5] In 2010, Twitter fell victim to a Cross Site Scripting attack whereby tweets themselves could be script injected and executed in a viewer's browser[6] A few years later in 2012, Google's GMail was the subject of a stored XSS attack, where malicious code stored as user info on Google+ would be rendered completely unsanitized in the GMail app.[7] In 2013, Facebook encountered CSRF bugs that allowed attackers to takeover user accounts.[8] And these are just the well-known companies. Across the tech sector, front-end security flaws open up XSS holes and entry-points for a diverse slew of technical incursions. So the reason that front-end security is so important becomes simple: a wall is only as strong as its weakest point. Even with the tightest server-side security, attackers can wreak havoc through simple front-end gaps. An attacker doesn't need to break into a database if he or she can simply impersonate a user, and she doesn't need to send phishing emails if she can trick a web app into sending all of the user's data itself!

One must also consider that most web applications are based on user-input. Think of any app with a large user-base and review the countless ways in which a user can input information to the system. Each of these spots, if not correctly protected, is a potential security risk. On the other side of the coin, take a look at any view of the app where user data (that is not your own) is rendered. These too, if not sanitized or character-escaped properly could result in malicious code being rendered and executed in the victim's browser (similarly to the GMail vulnerability). In the world of static HTML and large protected frameworks, developers did not have to worry so intensely about these issues, but the advent of independent Javascript frameworks and Javascript heavy front-end development places an extreme importance on these gaps in security. So as front-end developers, we are placed in a quite dire situation: one in which we are no longer fully protected by the frameworks we use and in which we must write application code that protects itself from these types of attacks. While lots of small frameworks make front-end development more accessible than ever, this framework decentralization is making security harder than ever before.

---

[5] "Front End Security is a thing, and you should be concerned ..." 2014. 15 Dec. 2015 <https://web-design-weekly.com/2014/07/09/front-end-security-thing-concerned/>
[6] "All about the "onMouseOver" incident | Twitter Blogs." 2015. 15 Dec. 2015 <https://blog.twitter.com/2010/all-about-the-onmouseover-incident>
[7] "Google Mail Hacking - Gmail Stored XSS - Ben Hayak." 2013. 15 Dec. 2015 <http://www.benhayak.com/2012/06/google-mail-hacking-gmail-stored-xss.html>
[8] "Facebook CSRF leading to full account takeover (fixed) - Pyxio." 2013. 15 Dec. 2015 <http://pyx.io/blog/facebook-csrf-leading-to-full-account-takeover>

IV. Action items/Defenses

Now that the magnitude of front-end security is apparent, it is important to understand exactly what kinds of issues arise when we do not take security into consideration during the app development process. There are two principal issues we have mentioned so far: Cross Site Scripting and Cross Site Request Forgery, both of which appear on the OWASP Top 10 list of common security flaws.[9] These security vulnerabilities can result in attacks that hurt users by stealing their personal information, execute unauthorized actions while impersonating a user, break the application partially or entirely, and represent a false image of the application's makers or users.

As mentioned before, Cross Site Scripting vulnerabilities are the result of several security holes. On the input side, it can occur when user-provided data is not sanitized and is stored unfiltered to be returned to users on page renders. While this is not in and of itself the worst vulnerability, if untrusted user data is neither sanitized on input before storage in the database nor on page render, XSS is a serious concern. Another point to consider is that often data is inserted not only as plaintext but also as DOM element attribute names, DOM element attribute values, image urls, page styles, HTML tags, etc., meaning that each contexted for DOM insertion must be addressed individually, while certain types of DOM insertion must be avoided entirely. OWASP's "Cross Site Scripting Prevention Cheat Sheet" outlines the steps necessary to comprehensively prevent XSS, and the list is not short. Any one of the defense items outlined (these are publicly available to attackers as well, of course) is a security gap that must be addressed. Luckily, there are extant templating frameworks with rather large communities that address some of these issues, such as Handlebars.[10] But in situations where database info is used across several sub-applications, standardized libraries or security measures must be taken vigilantly across all portions of the app, or else corrupt data that may be prevented from rendering in one portion of the app would not be in others (for example if the templating library is only used for initial renders and calls to jQuery's .hmtl() method, which does not escape HTML, are used for page updates). Overall, it's very clear that preventing XSS on the front end is a difficult, albeit mostly achievable goal: one that can and should be implemented wherever possible.

A related type of attack that has been mentioned is Cross Site Request Forgery, where attackers make unauthorized (mostly data-altering) requests to a service using a user's credentials.[11] While CSRF can happen without XSS vulnerabilities, holes opened by XSS can lead to access of user credentials stored in the browser. While defending against XSS surely diminishes CSRF risk, it cannot prevent these attacks on its own. Specifically because of the large separation between front and back end environments in modern web applications, the browser must exchange authentication and session data with the back end; it is this behavior that hijackers can exploit. OWASP lists a few frameworks

---

[9] "OWASP Top Ten Cheat Sheet - OWASP." 2012. 15 Dec. 2015 <https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet>

[10] "wycats/handlebars.js · GitHub." 2011. 15 Dec. 2015 <https://github.com/wycats/handlebars.js/>

[11] "Cross-Site Request Forgery (CSRF) - OWASP." 2006. 15 Dec. 2015 <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>

(including Ruby on Rails) that have built in defenses against this sort of attack, but the front-end frameworks we have come to know and love are absent this list. Defense against this sort of attack requires first fixing all XSS holes. Once XSS is prevented, a developer must first implement a "Synchronizer Token Pattern" that protects the aforementioned transmissions between the front and the back end of the application. This method uses a challenge token embedded in the actual HTML of the form, meaning that an attacker would essentially have to guess the token and send it along with the session info in a request to the application. The use of a challenge token helps mitigate CSRF vulnerabilities.[12]

It becomes very quickly apparent that securing against these types of attacks is no small undertaking, but not doing so can lead to dastrous effects, ranging from cute message popups to comprehensive user account takeovers. By addressing some of these attacks on the front end through certain sanitization methods, you can aggressively prevent many XSS security holes. As part of this paper, I have built a lightweight input sanitization framework: Input Sentinel. Input Sentinel allows a developer to use simple regular expressions to sanitize user input by watching the page for dynamic updates and observing form submissions. When forms are submitted, the inputs from the form are validated against the current Sentinels (regexes), and the form submission is prevented (by default) if a match is found. A callback can be set up to react to malicious input attempts. The source code for this project can be found here, where an example of the framework in action can be viewed.

V. Conclusion

As of recently, we have seen a large influx of small, front-end Javascript frameworks that promote less opinionated web development. While making app creation more accessible to the masses, they enforce fewer security measures than larger frameworks like Rails does, and due to flexible nature of Javascript as a programming language, allow developers to cut corners when it comes to security. As a result, front-end security is in a more dire position than ever, often requiring developers themselves to address major security concerns such as XSS and CSRF. The difficulty of securing a front-end application becomes magnified by newer architecture trends such as dynamic page updates and single-page apps. Cross Site Scripting can cause massive damage to both users and a company's reputation and reliability, and Cross Site Request Forgery, which can result from XSS holes, leads to stolen user information, damaged user representation, and even complete account takeover in some cases. While securing against XSS and CSRF requires a large amount of defense programming, not doing so can result in even more work in the form of remediation. Even large (and naievely assumed secure) companies such as Facebook, Google, and Twitter have seen the devastating efficacy of XSS and CSRF attacks. It is my hope that this paper inspires a sense of duty in front-end developers to

[12] "Cross-Site Request Forgery (CSRF) Prevention ... - owasp." 2010. 15 Dec. 2015 <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet>

address the security concerns listed herein and promote best practices when programming applications. I have begun an open-source, easy to use sanitization tool that I hope will contribute to bettering the current state of front-end security.

VI. Bibliography

The following sources were used to research the topics discussed in this paper

"All About the 'OnMouseOver' Incident | Twitter Blogs." *All about the "onMouseOver" incident | Twitter Blogs*. Web. 15 Dec. 2015. <https://blog.twitter.com/2010/all-about-the-onmouseover-incident>

"Ben Hayak - Security Blog: Google Mail Hacking - Gmail Stored XSS - 2012!" *Ben Hayak - Security Blog: Google Mail Hacking - Gmail Stored XSS - 2012!* Web. 15 Dec. 2015. <http://www.benhayak.com/2012/06/google-mail-hacking-gmail-stored-xss.html>

"Cross-Site Request Forgery (CSRF)." - *OWASP*. Web. 15 Dec. 2015. <https://www.owasp.org/index.php/cross-site_request_forgery_(csrf)>

"Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet." - *OWASP*. Web. 15 Dec. 2015. <https://www.owasp.org/index.php/csrf_prevention_cheat_sheet>

"Cross-Site Scripting (XSS)." - *OWASP*. Web. 15 Dec. 2015. <https://www.owasp.org/index.php/cross-site_scripting_(xss)>

"Documentation." *Security in Django*. Web. 15 Dec. 2015. <https://docs.djangoproject.com/en/1.9/topics/security/>

"Front End Security Is a Thing, and You Should Be Concerned about It - Web Design Weekly." *Web Design Weekly*. N.p., Sep. 2014. Web. 15 Dec. 2015. <https://web-design-weekly.com/2014/07/09/front-end-security-thing-concerned/>

"Front-End JavaScript Frameworks." *GitHub*. N.p., 2014. Web. 15 Dec. 2015. <https://github.com/showcases/front-end-javascript-frameworks>

"Module ActionView::Helpers::SanitizeHelper." *ActionView::Helpers::SanitizeHelper*. Web. 15 Dec. 2015. <http://api.rubyonrails.org/classes/actionview/helpers/sanitizehelper.html>

"MutationObserver." *Mozilla Developer Network*. Web. 15 Dec. 2015. <https://developer.mozilla.org/en-us/docs/web/api/mutationobserver>

"Pyxio." *Facebook CSRF leading to full account takeover (fixed)*. Web. 15 Dec. 2015. <http://pyx.io/blog/facebook-csrf-leading-to-full-account-takeover>

"Static Vs. Dynamic Websites." *Code Conquest*. Web. 15 Dec. 2015. <http://www.codeconquest.com/website/static-vs-dynamic-websites/>

"Wycats/Handlebars.Js." *GitHub*. Web. 15 Dec. 2015. <https://github.com/wycats/handlebars.js>