Analysis of Security Vulnerabilities in JavaScript Web Applications

David Lyle

david.lyle@tufts.edu

Mentor: Ming Chow

Abstract

Attacks on JavaScript vulnerabilities in web applications are among the easiest to implement and the most devastating. A malicious user does not need to do complex analysis or in depth hacking to exploit holes in application security. Usually they just need to take advantage of the trust between the user and the browser. The simplicity of these attacks makes them that much more dangerous. A creative attacker can use weaknesses to steal information, tamper with accounts, spread malware or commit fraud. This is a problem that can easily be fixed by educating programmers and employing good programming style and structure. Despite the risk involved and the simple solution, JavaScript programmers, especially beginners, often overlook these vulnerabilities. This paper will examine the different weaknesses associated with JavaScript Web Applications, and outline how these issues can be avoided.

Introduction

One of the major trends in software over the past 10 years has been an increase in using the web to provide services. Cloud based applications are extremely popular, and it has become a necessity for successful companies to have a strong web presence. This spike in web activity has made JavaScript extremely popular and important. Today JavaScript is the 11th most popular programming language in the world. Used to make web pages dynamic, JavaScript allows features such as animation, and user interactivity. The vast majority of web sites rely on JavaScript to enhance for their core functionality. With the wide spread use of JavaScript, it is more important than ever that it is used safely. Unfortunately this is too often not the case.

To the Community

Although it is important for everyone to understand the dangers of using the Internet, this paper is more geared towards programmers. The safety of the web and its users ultimately rests in our hands. I chose this topic because I think that it is especially important that the programming community is educated about this issue. The Internet is a tool that has become closely integrated into our everyday lives, and it is only becoming more crucial. We have a responsibility to make the Internet as safe as possible for the public. Not only does this benefit society, but it also helps your site if it is well known to be trustworthy.

Once you know what mistakes lead to vulnerabilities in web applications it is easy to avoid them. Simply understanding the problem and spreading awareness will significantly decrease the number of vulnerabilities on the web. The importance of education is especially true for novice

programmers. Coders who are just beginning to learn how to build a web application need to learn to do it the right way. If they develop bad habits, it will become much harder to change the tide. It is an unfortunate fact that many tutorials for web frameworks fail to emphasize security. It is not enough to provide the features that ensure security. You also need to stress that these features should be used by putting them in examples and tutorials. If the community at large doesn't decide to make security an essential part of building applications, just knowing how to mitigate vulnerabilities won't make a difference.

Action Points

The following points will elaborate on well known vulnerabilities in JavaScript web applications. These Open Web Application Security Project (OWASP) consistently includes these vulnerabilities in the top ten risks facing applications each year. They can also both be prevented pretty easily using techniques that I will lay out.

1. Cross Site Scripting

Cross Site Scripting, abbreviated XSS, is the most common type of attack on web applications. It consists of a malicious user embedding scripts in a page. These scripts would then execute in the browser when a user loads the page or triggers an event. This attack takes advantage of trust between the application and the user. The application incorrectly trusts the user to treat the application nicely. Most often this vulnerability is exposed when the user is allowed to input content that is then dynamically displayed on the page. All an attacker has to do is input a script instead of the expected input. In HTML the <script></script> tag is used to enclose JavaScript that the page is supposed to execute. If the user inputs JavaScript code within these script tags, the browser cannot distinguish between this input and scripts that the application is intended to run. The code will execute and this gives the attacker an immense amount of power. The most benign XSS attacks will just play with the Document Object Model to glean information about the application or change the appearance of the page. In most cases the impact will be much worse. One of the most powerful things about an XSS attack is that the code will execute in the user's browser. Therefore if a user is logged in, the script will be able to execute as a logged in user. This allows the attacker to bypass security and affect a much more deadly attack. If you fall victim to an XSS attack the cracker can hijack your attack, spread malware to your computer, view your browser history, and even control your browser remotely. Obviously this an immense amount of power. Not all XSS attacks will be this severe, but it is possible. If an attacker finds an XSS vulnerability, it is incredibly easy to exploit. Once they have control of your browser they are really only limited by their own ingenuity. Some of the more subtle, but equally

dangerous attacks are just aimed at stealing data. The attacker can steal information about the application and the user. Using this information they may be able to mount another more sophisticated attack. Clearly the dangers of Cross Site Scripting cannot be overstated. This vulnerability can be easily avoided, and yet it remains one of the most powerful and popular attacks.

The methods of preventing Cross Site Scripting attacks all deal with sanitizing user input before it is served to the page. You need to make sure that there is no JavaScript or recognizable HTML in the user input. One way to do this is called escaping. This process goes through the input and replaces special characters with a harmless character encoding. By this process <script> would become <script>. The encodings < and > correspond to less than and greater than respectively. This process renders JavaScript input inert. There are several easy ways to escape user input using different JavaScript libraries. Underscore.js contains many useful JavaScript functions including _.escape(). The popular web framework Backbone.js uses Underscore.js as a dependency, and therefore also has escape capabilities. The models in Backbone have a built in escape method. Using this you can escape user input before it is persisted to the database or loaded into the page. Underscore templates also contain functionality for escaping content. Once again this feature is also available in the Backbone framework. When you wish to interpolate a variable in a template, you can use the standard tags <%= … %>, or you can use <%- … %> which automatically escapes the value. These are important features which should definitely be taken advantage of in any application using Backbone or Underscore. Another way to escape content when it is being rendered is using jQuery's text(). When inserting content into a DOM element you can use $(“element”).html(...) or $(“element”).text(...). The former will treat the value passed as HTML while the latter treats it as escaped text. Simply by adding a few new methods to your program you can secure your application from the threat of XSS attacks. It is very important that this becomes common practice for all web developers.

2. Cross Site Request Forgery

Cross Site Request Forgery, or XSRF, is not as common as Cross Site Scripting, but it is still a very dangerous attack. For this attack to work the user must first be logged into a web application. Then the overly trusting user is tricked into making a request from another site to the targeted web application. Since the user is logged in, the malicious request has access to the user's account. They can then do whatever they want with the account. Unlike Cross Site Scripting, Cross Site Request Forgery exploits the trust that the application has in the user's identity. The classic example is when a user is logged into their on line bank. If this on line bank has an XSRF vulnerability, then an attacker can target it and make a request to transfer money from the victim's account to their own. Once again this

attack is simple yet dangerous. Any XSRF exploit involves more subtle trickery than complicated code. You need to target a user who will be logged into the target application. Often attacker's will find someone who uses the web site that they are targeting, and send them an email. Inside the email the attacker will encode some way of sending the request to the target application. One of the cleverer ways to do this is to put the application URL as the source of a zero-byte image. The image will fail to load, but the request will be sent. The beauty of this is that the victim has no indication that a malicious request has been sent except for the failed picture load. Although the user is the one who triggers the request from a suspicious site, the fault for an XSRF vulnerability lies with the application. If the application managed authentication correctly, a suspicious site would not be able to make requests to the user's account.

The way to properly manage authentication is to generate unique tokens for every user session. These tokens are then appended to the requests from the user. The application can verify that the token matches the token given to the user before authenticating requests. This relies on the token being secret from anyone except the user. Otherwise the malicious site could just copy the token and bypass the security. This technique is slightly more complicated than the solution to Cross Site Scripting, but it is very effective. Once you implement this once it is very easy to replicate. Another way to mitigate XSRF vulnerabilities is through Cross Origin Resource Sharing (CORS). This is a security measure that most browsers include today. Basically it regulates whether or not one site can access the resources at a different site. However, CORS cannot be relied upon because it varies depending on the browser and does not completely block requests. To really prevent XSRF vulnerabilities, programmers should get in the habit of using tokens for applications with any sort of user authentication.

Conclusion

JavaScript web applications provide the advantage of dynamic content and fast responsiveness. And in today's world they are an exciting and important tool in software. This is not going to change anytime soon. In fact web applications are only getting more and more prevalent. In order for this software to be effective and safe, we need to impose a new set of habits when it comes to designing web applications. Security should be a top priority. JavaScript is by nature insecure, because it is transparent to users and run in the browser. Even more importantly it opens up applications to Cross Site Scripting and Cross Site Request Forgery attacks. Both of these are extremely popular and dangerous vulnerabilities that need to be prevented. Coding applications that are secure against these attacks is not the difficult part. The difficult part is making the programming community care about these vulnerabilities. If web developers realize the severity of these issues then they will start to

program with more care. It is even more important that new developers learn to program with good secure style. Too many web application tutorials and frameworks completely ignore security. In order to make web applications safe, the next generation of programmers needs to develop good habits. Eventually if this change becomes widespread, Cross Site Scripting and Cross Site Request Forgery exploits will no longer be the most common web application vulnerabilities.

References

"Cross-Site Request Forgery." https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF). November 9, 2013.

Glynn Fergal. "JavaScript Security." http://www.veracode.com/security/javascript-security.

Glynn Fergal. "Cross-Site Scripting (XSS) Tutorial: Learn About XSS Vulnerabilities, XSS Injections and How to Prevent Cross Site Scripting Attacks." http://www.veracode.com/security/xss.

Glynn Fergal. "Cross-Site Request Forgery Guide: Learn All About CSRF Attacks and CSRF Protection." http://www.veracode.com/security/csrf.

Khosravi, Daniel. "Advanced Security In Backbone Application." http://danialk.github.io/blog/2013/07/28/advanced-security-in-backbone-application/. July 28, 2013

Thompson, Steven A. "Securing JavaScript Web Apps." http://blog.sathomas.me/post/securing-javascript-web-apps. April 9, 2013.