shortcut

Your Short Cut to Knowledge

Secure ASP.NET AJAX Development

Jason Schmitt

Addison-Wesley
Pearson Education

www.awprofessional.com

SECTION 2

AJAX Security Pitfalls

We have discussed up to this point how AJAX technologies basically represent a new way of doing things in Web applications using some relatively old Web technologies. The security of AJAX applications is a similar reversal of fortunes in that many developers will introduce old, familiar security vulnerabilities into their applications with new, unfamiliar development approaches. Let’s now turn to the security risks that you incur by choosing an AJAX approach in your Web development. The security risks discussed in this section apply to any AJAX implementation, not just the ASP.NET AJAX framework, so they are relevant to you no matter the technology you have chosen.

Before you continue, I must also warn you that this section does not provide an exhaustive list of Web application security vulnerabilities. We are focusing here on the small subset of application layer security vulnerabilities that AJAX commonly exposes you to, above and beyond the potential security vulnerabilities of traditional Web applications. You still need to understand the broader security picture and protect your applications from the start from all of the other Web application security risks that apply to your environment. For a refresher on the

Section 2:
AJAX Security Pitfalls

most common security vulnerabilities in Web applications and Web services, a couple of excellent free resources are the Open Web Application Security Project (OWASP) Top Ten[1] and the Microsoft MSDN Library "Improving Web Application Security: Threats and Countermeasures."[2]

Is AJAX Secure?

The first important point to remember is that AJAX in and of itself is not insecure. It is also not necessarily secure, either. It is the way you go about implementing AJAX that makes you secure or not.

The principle really applies to most Web technologies commonly in use today. For instance, there is nothing at all insecure about HTML. However, it is very easy to build a Web application replete with critical security vulnerabilities with nothing but HTML and a little bit of scripting. The security vulnerabilities in Web applications are rarely introduced by the technology alone, except in the case of flawed browser, Web server, or other such foundational software. Assuming that you are using fairly mainstream platform software that is fully patched and your network is effectively secured, the overwhelming majority of Web application security vulnerabilities are created by developers and sloppy programming. AJAX is no different.

AJAX is an approach for building Web applications, not a specific technology. As such, there are no security problems unique to AJAX, but a whole host of problems emerge from misuse of the underlying technologies. For instance, one of the most dangerous emerging threats to AJAX applications is cross-site request forgery. Cross-site request forgery, which will be discussed in detail later, is entirely possible using even older technologies such as IFRAMEs and image tags. The reason AJAX security is so important now is that it is just making it easier for the attacker to accomplish complex attacks and a lot harder for the user to detect them.

[1] *OWASP Top Ten: www.owasp.org.*

[2] *Improving Web Application Security: http://msdn2.microsoft.com/ en-us/library/ ms994921.aspx.*

As we discussed before, AJAX is a group of technologies—JavaScript, `XMLHttpRequest`, and XML—that have historically been well understood in isolation by specialized developers, but rarely when used extensively together in a rich client environment such as what we are seeing now. In fact, `XMLHttpRequest` is pretty restrictive and safe to use. None of these technologies is especially insecure by itself, but you can certainly do very risky and vulnerable things with each of them. When you combine them with so many other basic technologies—such as dynamic HTML (DHTML), Document Object Model (DOM), Web services, Java/C#/Visual Basic, and PHP, to name just a few—to create a highly complex, interactive Web application, the degree to which developers can comprehensively understand and address security in all of these technologies decreases.

AJAX Security Pitfalls

Many of the Web applications you have developed up until now might have been very secure, but the bottom line is that you are increasing your security risk by adding AJAX concepts to an existing application or by developing an AJAX application from scratch. This does not mean that you should shy away from AJAX if you intend to build a secure application. It just means that you need to take extra care to understand the issues at hand. In this section, you'll learn about all of the security mistakes that are common with AJAX applications. Most of these security vulnerabilities exist with traditional Web applications, so we'll describe the nature of the security pitfall and then discuss why it is of particular concern with AJAX. In general, AJAX just makes it much easier to make these mistakes. But rest assured; secure AJAX is easy if you and your team are aware of these pitfalls and follow the few simple, secure AJAX principles we discuss in Section 3.

Increased Attack Surface

The first way in which you add risk to your applications by taking an AJAX approach is by increasing the attack surface complexity of the application as compared to a traditional Web application. Consider a traditional Web application that you have been working on recently and think of all of the ways in which someone might be able to attack it. Think of every input to every page, each communication between the client and server, each connection between business logic and your database. All of these points where information is provided to your application or transmitted from one component to another are potential points of vulnerability because an attacker can either provide malicious or spoofed information or capture information as you are moving around. All of these points of vulnerability comprise your application's attack surface. Figure 2-1 shows a typical Web application attack surface.

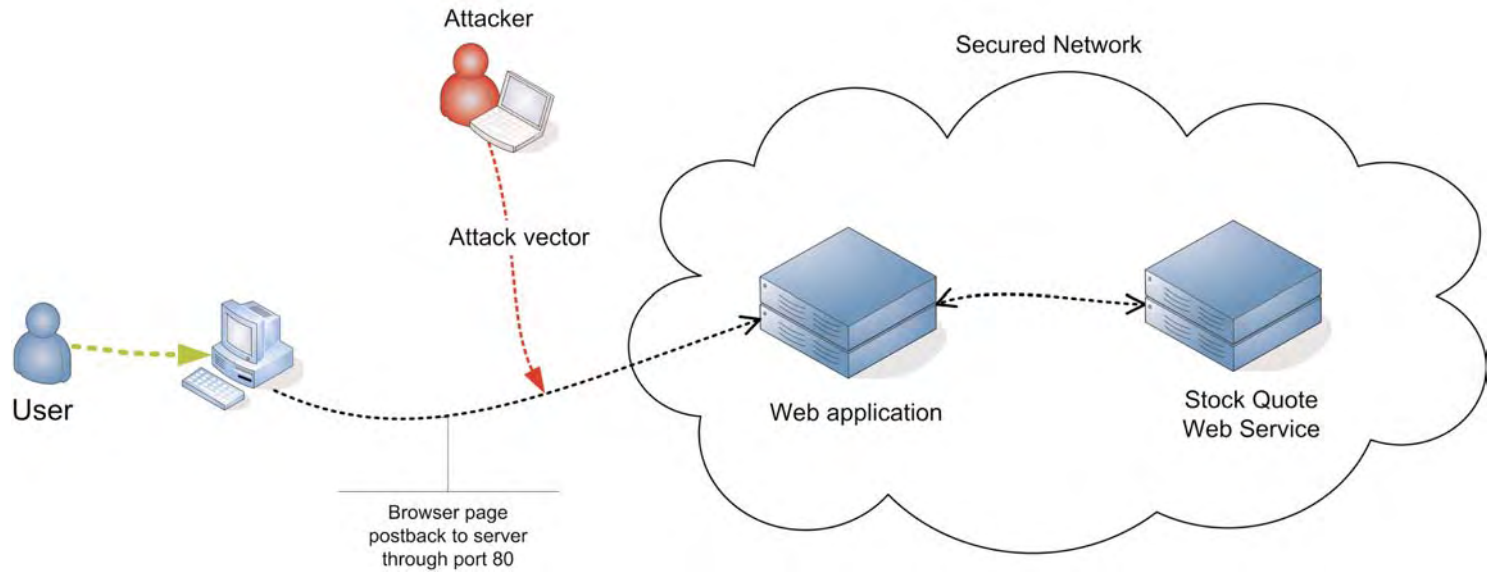


FIGURE 2-1
Simple Web Application Attack Surface

When threat modeling your application during the design process, you must first define this application attack surface to understand the superset of all potential vulnerability points. You have to be open-minded and consider all of them no matter how difficult you might think it would be to compromise one of them. Some of them are obviously easier to defend than others, but your job as a developer is to ensure that no vulnerabilities exist in your code at any of these points. So naturally, the more potential avenues of attack that you have, the more difficult it is to defend.

In most AJAX applications, you are able to deliver rich user interaction through frequent asynchronous calls from the client to the server, usually through Web services. For every new method you add to your services and every new call from your client back to them, you are adding new potential avenues of attack.

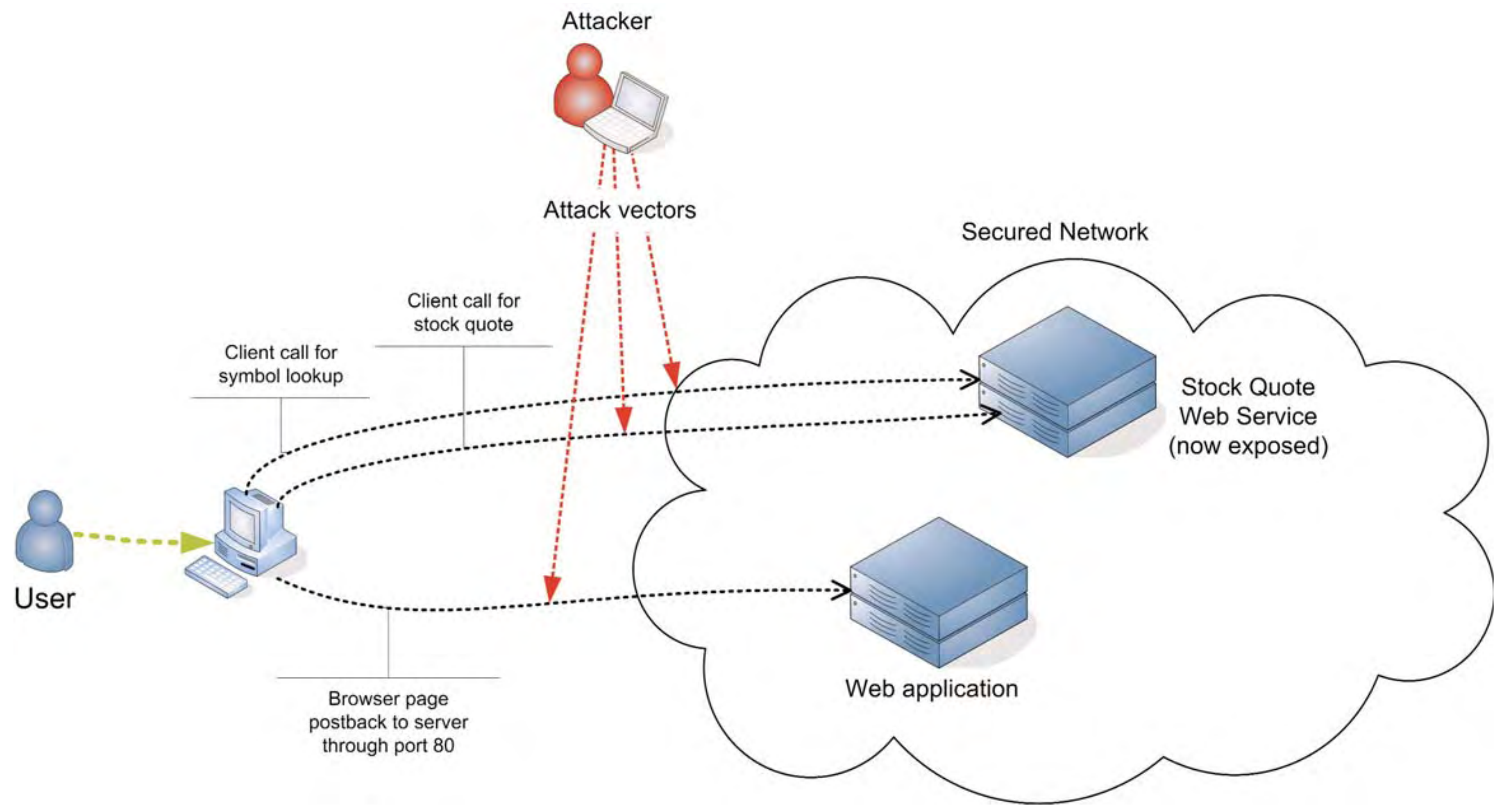
Even if it seems like the things you are adding to the application are simple or similar to other things you are already doing, every new avenue you open is still something else you have to consider and defend. With AJAX you are substantially increasing the attack surface of the application and potentially creating many new opportunities for security vulnerabilities. Even if you are not increasing the number of attack vectors to the application, you are definitely increasing the complexity of them. The road to securing AJAX is in ensuring that you treat these new avenues of attack with all the importance of everything else in your application.

Consider the stock quote example from Section 1. Without the ASP.NET AJAX implementation of the stock quote lookup that we described, that feature would have been possible only by posting the page back to the server with the stock symbol, the server making the necessary communications with the Web service, and the server returning the stock price back to you. In this traditional

Web application scenario, your exposed client is making a single postback to the server for processing and the server takes care of the rest, including input validation, Web service request and response handling, and preparation of the response back to the client browser. For this scenario, you can think of this version of the feature as having two avenues of attack: client to Web server and server application to Web service. Because you also own the Web service, the server application to Web service communications are purely internal and easy to secure, provided your network is protected. This leaves only one vulnerable input between the browser and server, which you can easily protect through input validation.

Now if you implement the same feature in ASP.NET AJAX as we did in Section 1, your attack surface is much more complicated. Figure 2-2 shows an increased attack surface after adding AJAX to the application. In this scenario, you have the client code talking directly to a Web service. You must now validate the input directly in the Web service because you are circumventing the server side of your application. You also must expose the Web service to the public network so that the client browser can call it directly. Your Web service also cannot tell if the request that is made of it is coming from the user's browser via the ASP.NET AJAX JavaScript proxy or if someone has hijacked the proxy and is making requests directly. Maybe you even added an auto-complete feature to the stock symbol field so that it suggests valid stock symbols based on the first two characters entered. This would require additional Web service methods and traffic that you need to worry about. To make this application feature more responsive in a fairly simple way, you have increased the complexity of the attack surface and increased the number of avenues of attack that you need to worry about by a factor of three. As the complexity of your applications increases, the security picture becomes much more difficult.

FIGURE 2-2
AJAX Application Attack Surface


Attacker
Attack vectors
Secured Network
Client call for stock quote
Client call for symbol lookup
Stock Quote Web Service (now exposed)
User
Web application
Browser page postback to server through port 80

Logic in the Client

Beefing up the client side of your application to make it more interactive and responsive often requires you to simply do more work in the client. The client either has to perform more business logic that you traditionally have been performing on the server, or has to decide when something is important enough to go to the server. The AJAX approach encourages you to make your client code perform more of these calculations and decisions in the browser, potentially enticing you to do sensitive things in an insecure manner within the browser. This manifests itself in a couple of ways.

First, you should remember that everything you do in client code is easily readable and understood by a user. No matter how you try to obscure your markup or client-side scripting, it is absolutely vulnerable to reverse engineering and manipulation—without exception. There is no way to effectively prevent this. By using the handy "View Source" feature of every browser, an attacker can easily read all of the source code of the client side of your application. If you link to script libraries in your page but they do not appear in the View Source for the page, these files are still readily downloadable from the server. Attackers can use all of this information to reconstruct the logic of your application, and increasingly they are simply stealing it to make malicious copies of your Web applications for use in phishing schemes.

Second, an attacker can fairly easily modify the behavior of all of the client code that you send to the browser. For many applications, modifying the source is harmless and only allows users to change the way the application looks or behaves in their browsers. On the other hand, everything that is sent to the browser, including all of the look and feel and JavaScript functions, is stored in the browser DOM. Attackers then have a variety of ways to manipulate every bit of information in the DOM to change the way the application behaves or to inject malicious behavior. Using the cross-site scripting techniques discussed later, attackers can essentially inject bad scripts into your pages and persist them as long as you allow and modify things in the DOM of your browser.

Let's again look at the simple ASP.NET AJAX stock quote example. If I am viewing this page in Internet Explorer, I can choose to View Source and see all of the local source code that was sent to me. I can use this source view to understand everything that the page is doing and even what the server might be expecting. If the page is using client-side validation, I can reverse engineer the JavaScript to understand what the developer considers to be valid input. From this I can tell whether the developer has missed something in his validation scheme, such as negative numbers, extremely large strings, and numbers that might allow me to crash the Web service. I can probe the server to see if the same validation exists there. The code in the browser gives me an excellent view of the application attack surface without much trouble at all.

I can also see what local variables were declared in the page to see if there is anything I can change locally to alter the behavior of the application. The source also tells me a little about the Web service that the page is communicating with. The ASP.NET AJAX client script code for communicating with the Web service is downloaded to the browser, some of which looks like this:

```
<script src="StockQuote.asmx/js" type="text/javascript">
</script>
```

From this script code, I can see the name and location of the Web service and how to get to the JavaScript proxy for the Web service. All of this is valuable information to a hacker that would not have been available had my server code been taking care of the communications with the Web service. Because it is necessary to put so much more information and logic in the client code that your application sends to the user's browser, you carry more of a burden to take security precautions in the server side of your application.