
Testing for Security in the Age of Ajax Programming

WHITE PAPER

1. Introduction

Ajax (Asynchronous Javascript and XML) allows a Web page to refresh a small portion of its data from a Web server, rather than being forced to reload and redraw the entire page as in traditional Web programming. Since they can make frequent, small updates, Web applications written with Ajax programming can present user interfaces that are more like desktop applications, which are more natural and intuitive interfaces for most users.

The flexibility and creativity that Ajax programming affords the developer also places a corresponding burden on him to ensure that his code is secure against malicious users and hackers performing SQL injection and similar attacks. Also, since delivering a secure application is part of delivering a quality application, the burden is probably felt even greater by the Quality Assurance (QA) team. The QA team will now need to develop an entirely new set of functional, performance and security testing methods in order to thoroughly test the quality of applications using Ajax programming against SQL injection attacks and other security concerns.

2. It's in the Code

Consider a hypothetical gourmet food e-commerce Web site. This site displays a map of the world to the user, and as the user navigates the mouse pointer over each country, the page uses Ajax programming to connect back to the Web server and retrieve a list of goods originating in that country. The following C# code snippet shows the Web method in which the database is queried:

```
[System.Web.Services.WebMethod] public System.Collections.IEnumerable  
GetProducts( string country)  
{  
    // update the select command to use the country parameter  
    this.SqlDataSource1.SelectCommand =  
        "SELECT * FROM [Product] WHERE Country = " + country + """;  
    // query the database and return the results  
    return this.SqlDataSource1.Select( DataSourceSelectArguments.Empty);  
}
```

The database query is being constructed on the fly with un-validated user input being sent directly to the database. This insecure programming technique creates a vulnerability to SQL injection attacks, which are potentially devastating to the Web application and its users. SQL injection vulnerabilities allow attackers to execute their own SQL queries and commands against the database, rather than those that the developers of the Web site intended. The entire database, including customer names, addresses, and credit card numbers, could be downloaded by such a command. The prices of the products could be modified. The entire database itself could be permanently deleted. Clearly, this is a very serious issue. If the developer fails to notice the problem, the next line of defense is the QA team.

The average developer will probably do a quick, cursory test of the application before passing it to the QA department, without checking thoroughly for SQL injection vulnerabilities or other important problems. Instead, he will mouse over a few countries on the map, check that the displayed results match those in the database, and then pass the code off. The average QA engineer typically will be much more thorough. He will mouse over every country on the map and check that the results match. He might even set up an automated test script that will mouse over every single pixel on the screen, and he will check to see if there are any errors in the Ajax programming or underlying page code.

But, even this extreme level of thoroughness won't be enough to find the SQL injection vulnerability. By using a Web browser (or automated script recorded from a Web browser) as his test tool, the tester has limited his potential requests to only those which the browser can send, and the browser is itself limited by the source code of the Web page. In the example above, the browser would be limited to sending only valid country parameters to the GetProducts method, since only valid countries are present in the page code. In other words, no matter where the user (or QA engineer) navigates with the mouse, the only parameters that would be sent are "GBR", "FRA", "USA", etc. Using only these valid, well-formed parameters will never reveal the SQL injection vulnerability. To do that, the QA team needs to expand their arsenal of test tools beyond browsers alone.

Since browsers are so limited, hackers generally don't use them to break into Web applications or execute their SQL injection attacks and other hacks. They use tools that operate at a much lower level, tools that are capable of sending raw HTTP requests to an address and displaying the raw HTTP response. Netcat is a popular tool for this purpose, as is telnet, which has the benefit of being installed by default with virtually every modern operating system. So, instead of opening a browser, navigating to a page, and viewing the rendered HTML response, the hacker types:

```
nc 172.16.60.250 80
GET /HTTP/1.0
and then receives the response:
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Wed, 10 May 2006 18:04:56 GMT
...
```

While interacting with Ajax programming may not seem like it would generate a round-trip request to the server since it doesn't refresh the entire page, it's doing exactly that. Like programming in standard hyperlink navigation or form submission, Ajax programming actions always have an HTTP request and response. So, armed with his low-level HTTP requestor tool, the hacker is now free to make attacks on the application that could never be possible with a browser alone. Instead of sending "GBR" or "FRA", he could send "XXX", or "!@#\$\$%", or "x' OR '1' = '1", which in this case would successfully exploit the SQL injection vulnerability.

3. Thinking like a Hacker

In order to successfully defend against the hacker using SQL injection or some other attack, the QA engineer has to think like the hacker. Since the hacker doesn't restrict himself to using just a browser to attack a Web application (with or without Ajax programming), neither should the QA engineer use just a browser to test it. At a minimum, the application should be tested with the same type of raw HTTP tool that the hacker uses. An even better approach is to use an automated security analysis tool that performs these tests. Automated tools can make thousands of test requests in an hour; work that would take a QA engineer a week or more to perform manually. Additionally, these tools generally have an extensive set of techniques that they use to detect security defects such as SQL injection vulnerabilities QA engineers would unlikely be aware of these techniques unless they had a background in information security. There are several excellent security analysis tools available commercially. Additional resources for learning about Web application security and security analysis tools include the Web Application Security Consortium (WASC), the Open Web Application Security Project (OWASP), and the SANS (SysAdmin, Audit, Network, Security) Institute.

4. Conclusion

It seems likely that Web applications using Ajax programming are the future of Web development. The robust user interface that Web pages comprised of Ajax programming can provide represents a huge leap in usability over traditional Web pages. But, this power comes with a price: the programmers and QA engineers must move beyond browsers alone when testing the application. Security vulnerabilities can lurk in code that is accessible only by specialized low-level request tools. Hackers will be more than willing to use these tools against your Web applications, so your QA team must use the same tactics to find the vulnerabilities first.