FREE CHAPTER
Books Available December 2007

AJAX SECURITY

BILLY HOFFMAN
BRYAN SULLIVAN

DRAFT MANUSCRIPT

Books Available

December 2007

This manuscript has been provided by Pearson Education at this early stage to create awareness for this upcoming book. **It has not been copyedited or proofread yet**; we trust that you will judge this book on technical merit, not on grammatical and punctuation errors that will be fixed at a later stage.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

All Pearson Education books are available at a discount for corporate bulk purchases. For information on bulk discounts, please call (800) 428-5531.

About the Authors

BILLY HOFFMAN is the lead security researcher for S.P.I. Dynamics, Inc. where he focuses on automated discovery of Web application vulnerabilities and crawling technologies. He has been a guest speaker at several high-level security and IT conferences and is well-known for his expertise in Ajax security. Billy wrote TinyDisk and is the creator of Stripe Snoop. In addition, Billy is involved in the Web Application Security Consortium (WASC) and spends his time contributing to OSS projects, writing articles, and giving presentations under the handle Acidus.

BRYAN SULLIVAN has been a professional software developer and development manager since graduating from Georgia Tech in 1995. For the last four years, he has been working in the application security sector of the software industry at S.P.I. Dynamics, Inc. While at SPI, he helped create the DevInspect® product, which analyzes Web applications for security vulnerabilities. He was involved in the creation of the Application Vulnerability Description Language (AVDL) and has three patents on security assessment and remediation methodologies pending review.

Chapter 6
Transparency in Ajax Applications

Myth: Ajax applications are black box systems, just like regular Web applications.

When most people use a microwave oven, they have no idea how it actually works. They only know that if they put food in and turn the oven on, the food will get hot in a few minutes. By contrast, a toaster is fairly easy to understand. When you're using a toaster, you can just look inside the slots to see the elements getting hot and toasting the bread.

A traditional Web application is like a microwave oven. Most users don't know how Web applications work and don't even care to know how they work. Furthermore, most users have no way to find out how a given application works even if they did care. Beyond the fundamentals, such as use of HTTP as a request protocol, there is no guaranteed way to determine the inner workings of a Web site. By contrast, an Ajax Web application is more like a toaster. While the average user may not be aware that the logic of the Ajax application is more exposed than that of the standard Web page, it is a simple matter for an advanced user (or an attacker) to "look inside the toaster slots" and gain knowledge about the internal workings of the application.

Black boxes vs. White boxes

Web applications (and microwave ovens) are examples of “black box” systems. From the user’s perspective, input goes into the system, and then output comes out of the system. The application logic that processes the input and returns the output is abstracted from the user and is invisible to him.

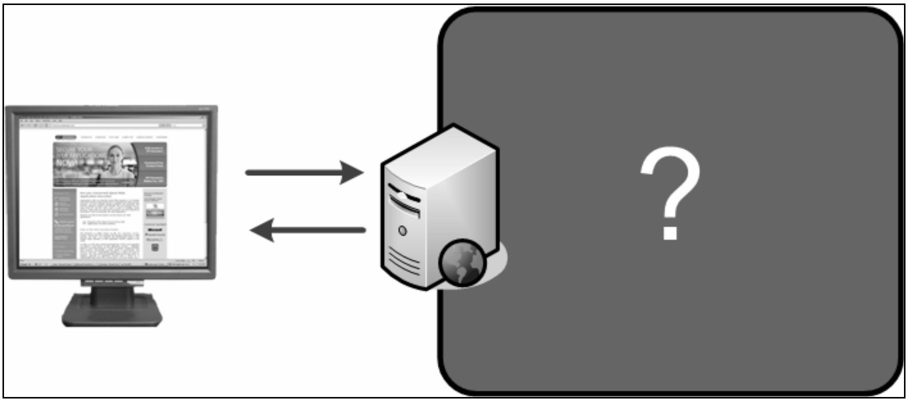


Figure 6-1

The inner workings of a black box system are unknown to the user

For example, consider a weather forecast Web site. A user enters his ZIP code into the application, and the application then tells him if the forecast calls for rain or sun. But how did the application gather that data? It may be that the application performs real-time analysis of current weather radar readings, or it may be that every morning a programmer watches the local television forecast and copies that into the system. Since the end user does not have access to the source code of the application, there is really no way for him to know.

> **Security Note**
>
> There are in fact some situations in which an end user may be able to obtain the application’s source code. These situations mostly arise from improper configuration of the Web server or insecure source code control techniques, such as storing backup files on production systems. Please review Chapter 3 for more information on these types of vulnerabilities.

“White box” systems behave in the opposite manner. Input goes into the system and output comes out of the system as before, but in this case the internal mechanisms (in the form of source code) are visible to the user.

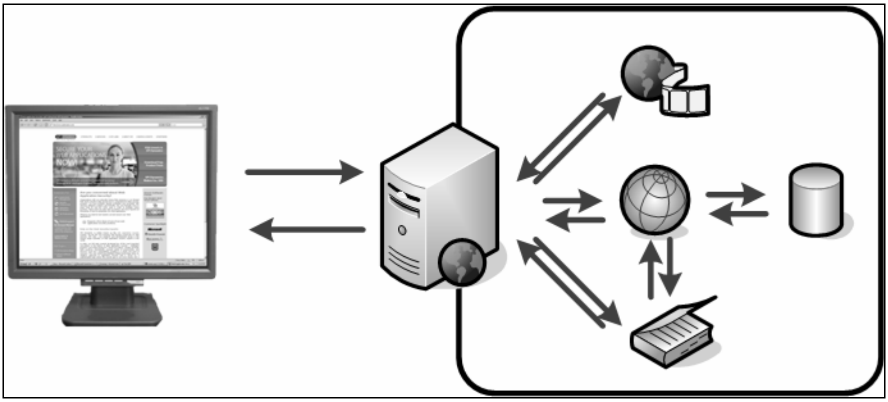


Figure 6-2
The user can see the inner workings of a white box system

Any interpreted script-based application, such as a batch file, macro, or (more to the point) a JavaScript application, can be considered a white box system. As we discussed in the previous chapter, JavaScript must be sent from the server to the client in its original, unencrypted source code form. It is a simple matter for a user to open this source code and see exactly what the application is doing.

It is true that Ajax applications are not completely white box systems since there is still a large portion of the application that executes on the server. However, they are much more transparent than traditional Web applications, and this transparency provides opportunities for hackers, as we will demonstrate over the course of the chapter.

It is possible to *obfuscate* JavaScript, but this is different than encryption. Encrypted code is impossible to read until the correct key is used to decrypt it, at which point it is readable by anyone. Encrypted code cannot be executed until it is decrypted. On the other hand, obfuscated code is still executable as-is. All the obfuscation process accomplishes is to make the code more difficult to read by a human. The key phrases here are that obfuscation makes code “more difficult” for a human to read, while encryption makes it “impossible”, or at least virtually impossible. Someone with enough time and patience could still reverse-engineer the obfuscated code. As we saw in Chapter 2, Eve created a program to de-obfuscate JavaScript. In actuality, the authors created this tool, and it only took a few days. For this reason, obfuscation should be considered more of a speed bump than a roadblock for a hacker: it may slow a determined attacker down but it will not stop him.

In general, white box systems are easier to attack than black box systems because their source code is more transparent. Remember that attackers thrive on information. A large percentage of the time that a hacker spends attacking a Web site is not actually spent sending malicious requests, but rather analyzing it to determine how it works. If the application freely provides details of its implementation, this task is greatly simplified. Let's continue the weather forecasting Web site example and evaluate it from an application logic transparency point of view.

Example: MyLocalWeatherForecast.com

First, let's look at a standard, non-Ajax version of MyLocalWeatherForecast.com:

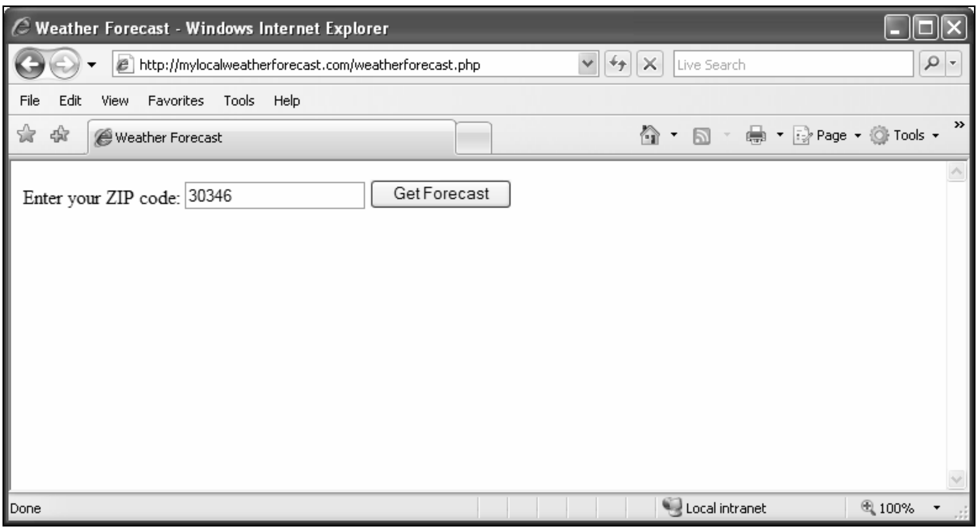


Figure 6-3
A standard, non-Ajax weather forecasting Web site

There's not much to see just from the rendered browser output, except that the server-side application code appears to be written in PHP, since the filename of the Web page ends in ".php". The next logical step an attacker would make would be to view the page source, so we will do the same.

```
<html>
  <head>
    <title>Weather Forecast</title>
  </head>
  <body>
    <form action="/weatherforecast.php" method="POST">
      <div>
        Enter your ZIP code:
        <input name="ZipCode" type="text" value=30346 />
        <input id="Button1" type="submit" value="Get Forecast" />
      </div>
    </form>
  </body>
</html>
```

There’s not much to see from the page source code either. We can tell that the page uses the HTTP POST method to post the user input back to itself for processing. As a final test, we will attach a network traffic analyzer (also known as a “sniffer”) and examine the raw response data from the server.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 18:23:12 GMT
Connection: close
Content-type: text/html
X-Powered-By: PHP/5.1.4

<html>
  <head>
    <title>Weather Forecast</title>
  </head>
  <body>
    <form action="/weatherforecast.php" method="POST">
      <div>
   Enter your ZIP code:
   <input name="ZipCode" type="text" value=30346 />
        <input id="Button1" type="submit" value="Get Forecast" />
        <br />
   The weather for December 17, 2006 for 30346 will be sunny.
      </div>
```

```
    </form>
  </body>
</html>
```

The HTTP request headers give us a little more information to work with. The header `X-Powered-By: PHP/5.1.4` confirms that the application is indeed using PHP for its server-side code, and additionally we now know which version of PHP is being used (5.1.4). We can also see from the `Server: Microsoft-IIS/5.1` header that Microsoft Internet Information Server (IIS) version 5.1 is being used as the Web server. Also, this implicitly tells us that Microsoft Windows XP Professional is being used as the server's operating system, since IIS 5.1 only runs on XP Professional.

So far, we have collected a modest amount of information regarding the weather forecast site. We know what programming language is used to develop the site, and the particular version of that language. We know which Web server and operating system are being used. These tidbits of data seem innocent enough – after all, what difference could it make to a hacker if he knew that a Web application was running on IIS versus Tomcat? The answer is simple: time. Once the hacker knows that a particular technology is being used, he can focus his efforts on cracking that piece of the application and avoid wasting time by attacking technologies he now knows to be unused. For example, knowing that XP Professional is being used as the operating system allows him to omit attacks that could only succeed against Solaris or Linux operating systems, and to concentrate on making attacks that are known to work against Windows. And if he doesn't know any Windows-specific attacks (or IIS-specific attacks, or PHP-specific attacks, etc,) it is a simple matter to find examples on the Internet.

Security Note

Disable HTTP response headers that reveal implementation or configuration details of your Web applications. The `Server` and `X-Powered-By` headers both reveal too much information to potential attackers and should be disabled. The process for disabling these headers varies among different Web servers and application frameworks; for example, Apache users can disable the `Server` header with a configuration setting, while IIS users can use the RemoveServerHeader feature of Microsoft's UrlScan Security Tool. This feature has also been integrated natively into IIS since version 6.

For maximum security, also remap your application's file extensions to custom types. It does little good to remove the `X-Powered-By: ASP.NET` header if your Web pages end in `.aspx` extensions. Hiding application details like these doesn't guarantee that your Web site won't be hacked, but it will make the attacker work that much harder to do it, and he might just give up and attack someone else.

Example: MyLocalWeatherForecast.com "Ajaxified"

Now that we have seen how much of the internal workings of a "black box" system can be uncovered, let's examine the same weather forecasting application after it has been converted to Ajax.

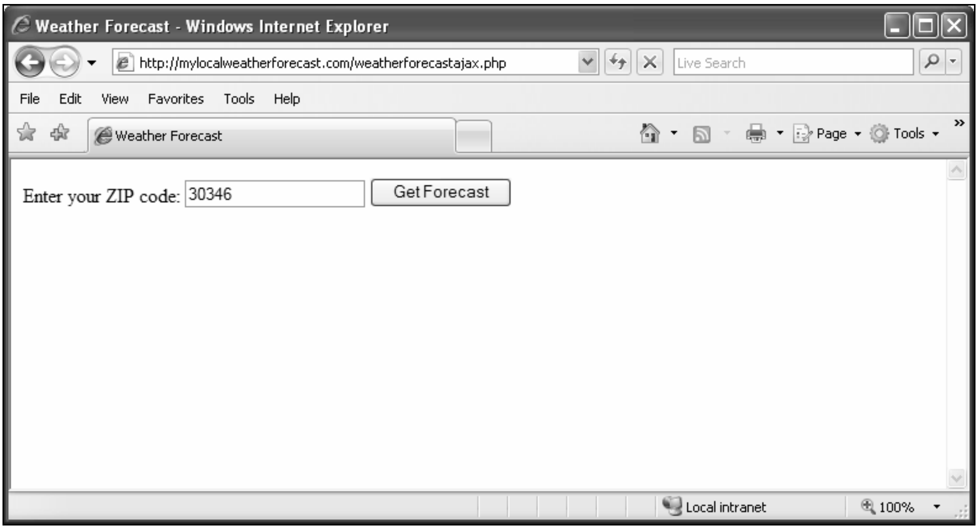


Figure 6-4
The Ajax-based weather forecast site

The new Web site looks the same as the old when viewed in the browser. We can still see that PHP is being used because of the file extension, but there is no new information yet. However, when we view the page source...

```
<html>
  <head>
  <script type="text/javascript">

    var httpRequest = getHttpRequest();

    function getRadarReading() {
      // access the web service to get the radar reading
      var zipCode = document.getElementById('ZipCode').value;
      httpRequest.open("GET",
        "weatherservice.asmx?op=GetRadarReading&zipCode=" + zipCode,
        true);
      httpRequest.onreadystatechange = handleReadingRetrieved;
      httpRequest.send(null);
    }

    function handleReadingRetrieved() {
      if (httpRequest.readyState == 4) {
        if (httpRequest.status == 200) {
          var radarData = httpRequest.responseText;
          // process the XML retrieved from the web service
          var xmldoc = parseXML(radarData);
          var weatherData =
            xmldoc.getElementsByTagName("WeatherData")[0];
          var cloudDensity = weatherData.getElementsByTagName
            ("CloudDensity")[0].firstChild.data;
          getForecast(cloudDensity);
        }
      }
    }

    function getForecast(cloudDensity) {
      httpRequest.open("GET",
        "forecast.php?cloudDensity=" + cloudDensity,
        true);
      httpRequest.onreadystatechange = handleForecastRetrieved;
      httpRequest.send(null);
    }

    function handleForecastRetrieved() {
      if (httpRequest.readyState == 4) {
        if (httpRequest.status == 200) {
          var chanceOfRain = httpRequest.responseText;
          var displayText;
          if (chanceOfRain >= 25) {
```

```
            displayText = “The forecast calls for rain.”;
          } else {
            displayText = “The forecast calls for sunny skies.”;
          }
        document.getElementById(‘Forecast’).innerHTML = displayText;
        }
      }
    }

    function parseXML(text) {
      if (typeof DOMParser != "undefined") {
        return (new DOMParser()).parseFromString(text,
          "application/xml");
      }
      else if (typeof ActiveXObject != "undefined") {
        var doc = new ActiveXObject("MSXML2.DOMDocument");
           doc.loadXML(text);
           return doc;
         }
       }

     </script>
   </head>
   </html>
```

Aha! Now we know exactly how the weather forecast is calculated. First, the function `getRadarReading` makes an asynchronous call to a Web service to obtain the current radar data for the given ZIP code. The radar data XML returned from the Web service is parsed apart (in the `handleReadingRetrieved` function) to find the “cloud density” reading. A second asynchronous call (`getForecast`) passes the cloud density value back to the server. Based on this cloud density reading, the server determines tomorrow’s chance of rain. Finally, the client displays the result to the user and suggests whether he should take an umbrella to work.

Just from viewing the client-side source code, we now have a much better understanding of the internal workings of the application. Let’s go one step further and “sniff” some of the network traffic.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 18:54:31 GMT
Connection: close
Content-type: text/html
X-Powered-By: PHP/5.1.4
```

```
<html>
  <head>
  <script type="text/javascript">
…
</html>
```

Sniffing the initial response from the main page didn't tell us anything that we didn't already know. We will leave the sniffer attached while we make an asynchronous request to the radar reading Web service. The server responds as follows:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 19:01:43 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 301

<?xml version="1.0" encoding="utf-8"?>
<WeatherData>
  <Latitude>33.76</Latitude>
  <Longitude>-84.4</Longitude>
  <CloudDensity>0</CloudDensity>
  <Temperature>54.2</Temperature>
  <Windchill>54.2</Windchill>
  <Humidity>0.83</Humidity>
  <DewPoint>49.0</DewPoint>
  <Visibility>4.0</Visibility>
</WeatherData>
```

This response gives us some new information about the Web service. We can tell from the `X-Powered-By` header that it uses ASP.NET, which might help an attacker as described earlier. More interestingly, we can also see from the response that much more data than just the cloud density reading is being retrieved. The current temperature, wind chill, humidity, and other weather data are being sent to the client. The client-side code is discarding these additional values, but they are still plainly visible to anyone with a network traffic analyzer.

Comparison Conclusions

Comparing the amount of information gathered on MyLocalWeatherForecast.com before and after its conversion to Ajax, we can see that the new Ajax-enabled site discloses everything that the old site did, as well as some additional items.

Information Disclosed	Non-Ajax	Ajax
Source code language	Yes	Yes
Web server	Yes	Yes
Server operating system	Yes	Yes
Additional subcomponents	No	Yes
Method signatures	No	Yes
Parameter data types	No	Yes

Table 6-1: Information disclosure in Ajax vs. Non-Ajax applications

The Web Application as an API

The effect of MyLocalWeatherForecast.com's shift to Ajax is that the client-side portion of the application (and by extension, the user) has more visibility into the server-side components. Before, the system functioned as a black box. Now, the box is becoming clearer; the processes are becoming more transparent.

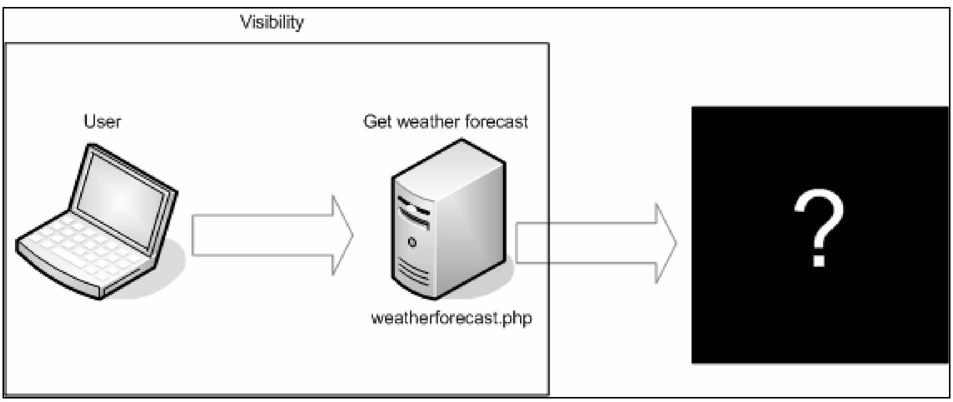


Figure 6-5
Client visibility of (non-Ajax) MyLocalWeatherForecast.com

Figure 6-5 shows the visibility of the old MyLocalWeatherForecast.com site. In a sense, MyLocalWeatherForecast.com is just an elaborate application programming interface (API). In the non-Ajax model, there is only one publicly exposed method in the API, “Get weather forecast”.

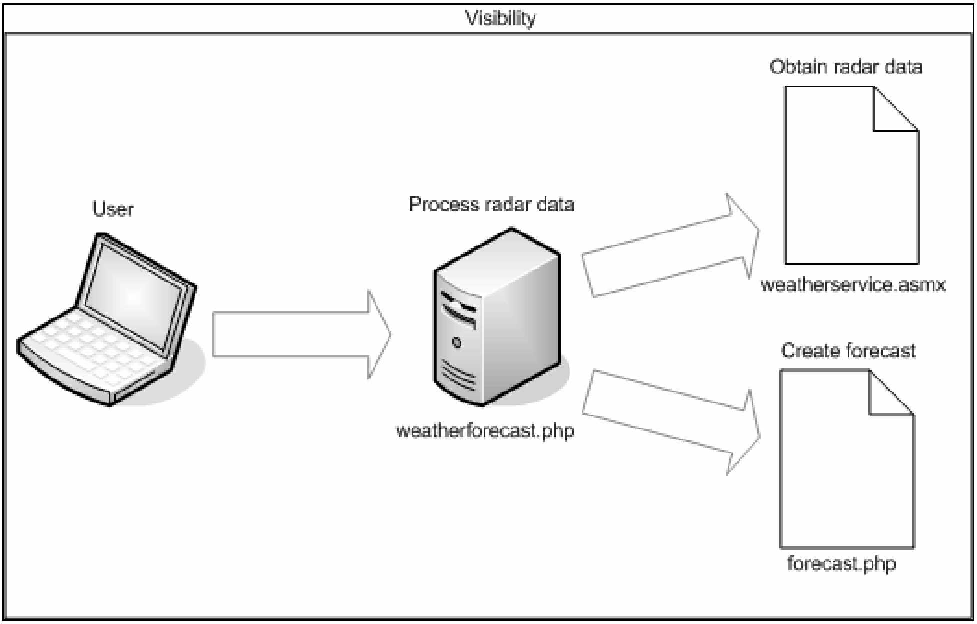


Figure 6-6
Client visibility of Ajax MyLocalWeatherForecast.com

Figure 6-6 shows the visibility of the new, Ajax-enabled MyLocalWeatherForecast.com. Not only did our API get a lot bigger (three methods instead of one), but its granularity increased as well. Instead of one, big “do it” function, we can see the individual subroutines that combine to calculate the result output. Furthermore, in many “real world” scenarios, the JavaScript client-side code is not defined in each individual page on an as-needed basis. Instead, all of the client-side JavaScript functions used on any page are collected into a single, monolithic script library which is then referenced by each page that uses it.

```
<script src=”ajaxlibrary.js”></script>
```

This architecture makes it easier for the site developers to maintain the code, since they now only have to make changes in a single place. It can save bandwidth as well, since a browser will download the entire library only once and then cache it for later use. Of course, the downside of this is that the entire API can now be exposed after only a single request from a user. The user basically asks the server, “Tell me everything you can do,” and the server answers with a list of actions. As a result, a potential hacker can now see a much larger attack surface, and additionally his task of analyzing the application is made much easier as well. The flow of data through the system is more evident, and data types and method signatures are also visible.

Data Types and Method Signatures

Knowing the arguments’ data types can be especially useful to an attacker. For example, if an attacker finds that a given parameter is an unsigned, 16-bit integer, he knows that valid values for that parameter range from 0 to 65,535 (2^{16}-1). However, the attacker is not constrained to send only valid values. And since the method arguments are sent as strings over the wire, the attacker is not even constrained to send valid data types. He may send a negative value, or a value greater than 65,535, to try to overflow or underflow the value. He may send a non-numeric value just to try to cause the server to generate an error message. Error messages returned from a Web server often contain sensitive information such as stack traces and lines of source code. Nothing makes analyzing an application easier than having its server-side source code!

It may be useful just to know which pieces of data are used to calculate results. For example, in MyLocalWeatherForecast.com, the forecast is determined solely from the current cloud density and not from any of the other current weather variables such as temperature or dew point. The usefulness of this information can vary from application to application. Knowing that the current humidity does not factor into the weather forecast at MyLocalWeatherForecast.com may not help a hacker penetrate the site, but knowing that a person’s employment history does not factor into a loan application decision at an online bank may.

Specific Security Mistakes

Beyond just the general danger of revealing application logic to potential attackers, there are specific mistakes that programmers make when writing client-side code that can open their applications to attack.

Improper Authorization

Let's return to MyLocalWeatherForecast.com. MyLocalWeatherForecast.com has an administration page, where site administrators can check usage statistics. The site requires administrative authorization rights in order to access this page, so that site users and other prying eyes will be prevented from viewing the sensitive content.

Since the site already used Ajax to retrieve the weather forecast data, the programmers continued this model and used Ajax to retrieve the administrative data: they added client-side JavaScript code that pulls the usage statistics from the server.

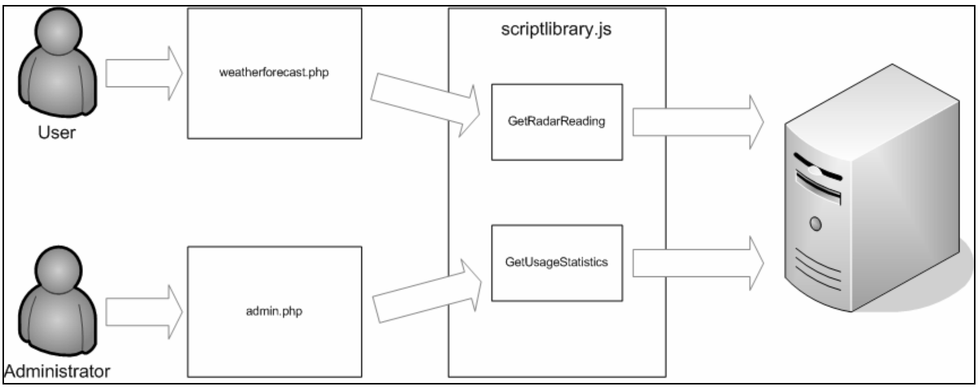


Figure 6-7
Intended usage of the Ajax administration functionality

Unfortunately, while the developers at MyLocalWeatherForecast.com were diligent about restricting access to the administration page (admin.php), they neglected to restrict access to the server API that provides the actual data to that page. While an attacker would be blocked from accessing admin.php, there is nothing to prevent him from calling the `GetUsageStatistics` function directly.

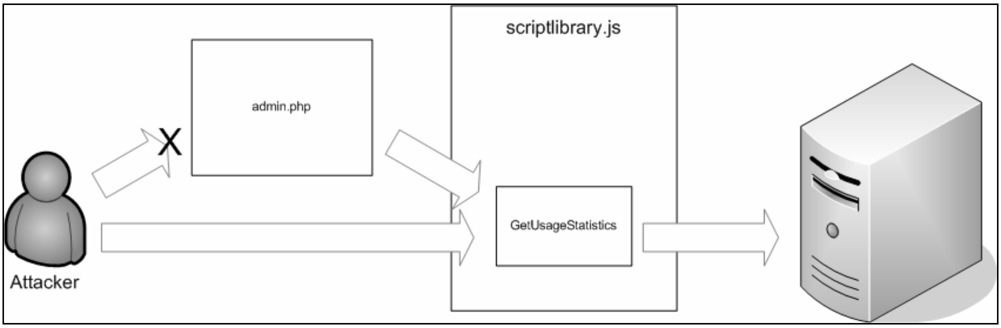


Figure 6-8
Hacking the administration functionality by directly accessing the client-side JavaScript function

There is no reason for the hacker to try to gain access to admin.php. He can dispense with the usual, tedious authorization bypass attacks like hijacking a legitimate user's session or guessing a username and password through brute force. Instead, he can simply ask the server for the administrative data without having to go to the administrative page, just like Eve did in her attack on HighTechVacations.net in Chapter 2. The programmers at MyLocalWeatherForecast.com never intended the `GetUsageStatistics` function to be called from any page besides admin.php. They might not have even realized that it could be called from any other page. Nevertheless, their application has been hacked and they are to blame.

Security Note

In this case, it was easy for the attacker to discover the `GetUsageStatistics` function and call it because it was defined in a shared library referenced by both the main user page weatherforecast.php and the administration page admin.php. However, even if `GetUsageStatistics` were to be removed from the shared library and defined only in admin.php, this would not change the fact that an attacker could still call the server method directly if he ever found out about its existence. Hiding the method is not a substitute for appropriate authorization.This is called relying on "security through obscurity" and is a dangerous approach to take. The problems with depending on obscurity will be discussed later in this chapter.

Some of the worst cases of improperly authorized API methods come from sites that were once standard Web applications but were later converted to Ajax-enabled applications. You must take care when "Ajaxifying" applications in order to avoid accidentally exposing sensitive or trusted server-side functionality. In one real world example of this, the developers of a Web framework made all of their user management functionality available through Ajax calls. But just like our fictional developers at MyLocalWeatherForecast.com, they neglected to add authorization to the server code. As a result, any attacker could easily add new users to the system, remove existing users, or change users' passwords at will.

Security Note

When converting an existing application to Ajax, remember to add authorization checking code to newly exposed methods. Functionality that was intended to be accessed only from certain pages will now be available everywhere. As a result, you can no longer rely on the authorization mechanisms of the page code. Each public method must now check a user's authorization.

Overly Granular Server API

The lack of proper authentication in the previous section is really just a specific case of a much broader and more dangerous problem: the overly granular server API. This problem occurs when programmers expose a server API and assume that the only consumers of that API will be the pages of their applications, and that those pages will always use that API in exactly the way that the programmers intended. The truth is, an attacker can easily manipulate the intended control flow of any client-side script code. Let's revisit the online music store example from Chapter 1:

```
function purchaseSong(username, password, songId) {

  // Note that the functions checkCredentials, getSongPrice,
  //   getAccountBalance, debitAccount, and downloadSong all
make Ajax
  //   requests back to the server. Their code is omitted
for brevity.

  // first authenticate the user
  var authenticated = checkCredentials(username, password);
```

```
  if (authenticated == false) {
    alert('The username or password is incorrect.');
    return;
  }

  // get the price of the song
  var songPrice = getSongPrice(songId);

  // make sure the user has enough money in his account
  if (getAccountBalance(username) < songPrice) {
    alert('You do not have enough money in your account.');
    return;
  }

  // debit the user's account
  debitAccount(username, songPrice);

  // start downloading the song to the client machine
  downloadSong(songId);
}
```

The intended control flow of this code is fairly straightforward: first the application checks the user’s username and password, then it retrieves the price of the selected song and makes sure the user has enough money in his account to purchase it. Next, the user’s account is debited for the appropriate amount, and finally the song is downloaded to his computer. All of this works fine for a legitimate user. But let’s think like our hacker Eve would, and attach a JavaScript debugger to the page to see what kind of havoc we can wreak.

We will start with the debugger Firebug for Firefox. Firebug will display the raw HTML, DOM object values, and any currently loaded script source code for the current page. It will also allow the user to place breakpoints on lines of script, as we do in Figure 6-9:

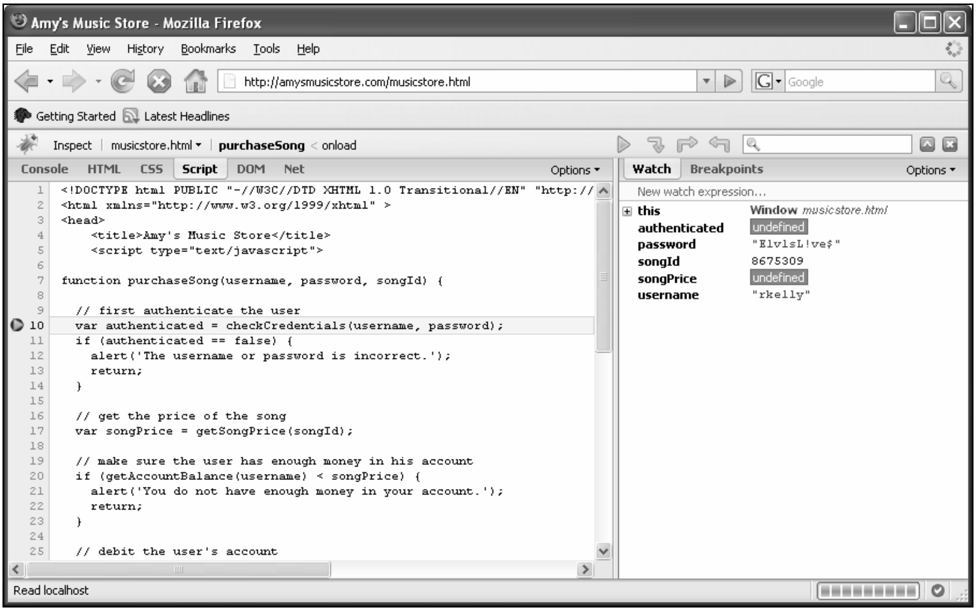


Figure 6-9
Attaching a breakpoint to JavaScript with Firebug

You can see that a breakpoint has been hit just before the call to the `checkCredentials` function. Let's step over this line, allow the client to call `checkCredentials`, and examine the return value.

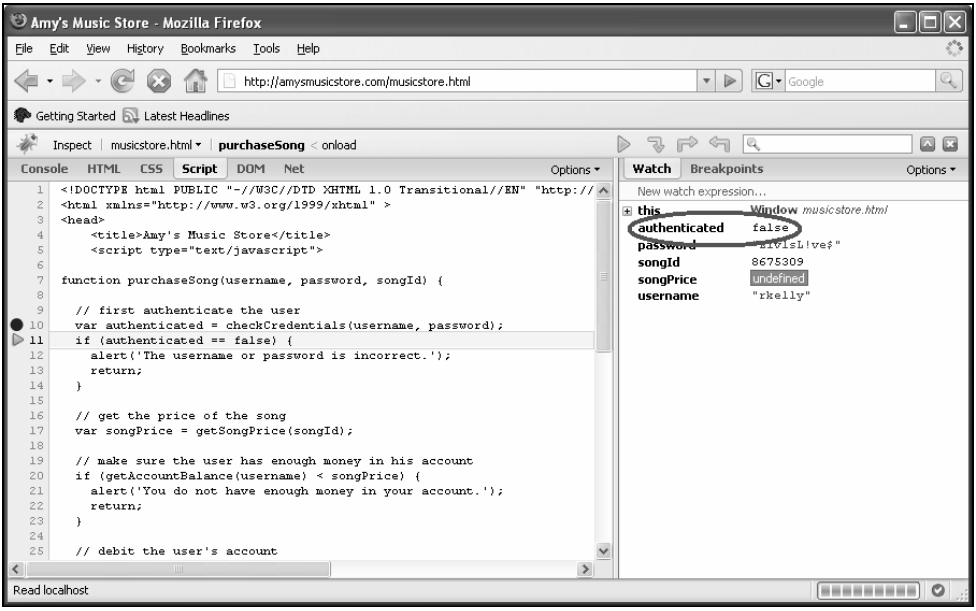


Figure 6-10

Examining the return value from checkCredentials

Unfortunately, the username and password that we provided do not appear to be valid. The value of the authenticated variable as returned from `checkCredentials` is false, and if we allow execution of this code to proceed as-is, the page will alert us that the credentials are invalid and then return from the `purchaseSong` function. However, as a hacker, this does us absolutely no good. Before we proceed, let's use Firebug to alter the value of authenticated from false to true:

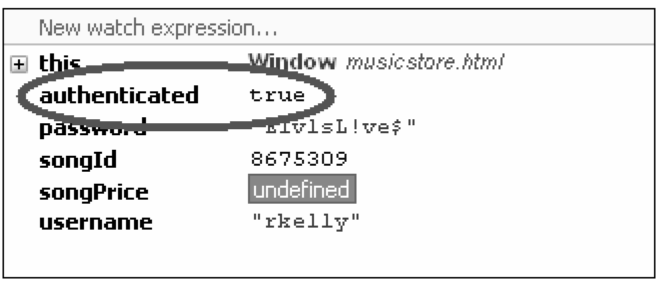


Figure 6-11

The attacker has modified the value of the authenticated variable from false to true

By editing the value of the variable, we have modified the intended control flow of the application. If we were to let the code continue execution at this point, it would assume (incorrectly) that we have a valid username and password, and proceed to retrieve the price of the selected song. However, while we have the black hat on, why should we stop at just bypassing authentication? We can use this exact same technique to modify the returned value of the song price, from $.99 to $.01 or free. Or, we could cut out the middleman and just use the Console window in Firebug to call the `downloadSong` function directly.

In this example, all of the required steps of the transaction – checking the user's credentials, ensuring that he had enough money in his account, debiting the account and downloading the song – should have been enforced to be performed as one single, atomic action. Instead of exposing all of these steps as individual methods in the server API, the programmers should have written a single `purchaseSong` method that would execute on the server, and that would enforce the individual steps to be called in the correct order with the correct parameter values. The exposure of overly-granular server APIs is one of the most critical security issues facing Ajax applications today. It bears repeating: never assume that client-side code will be executed the way you intend or even that it will be executed at all.

Sensitive Data Revealed to Users

Programmers often hardcode string values into their applications. This practice is usually frowned upon due to localization issues – for example, it is harder to translate an application into Spanish or Japanese if there are English words and sentences hardcoded throughout the source code. However, depending on the string values, there could be security implications as well. If the programmer has hardcoded a database connection string or authentication credentials into the application, then anyone with access to the source code now has credentials to the corresponding database or secure area of the application.

Programmers also frequently misuse sensitive strings by processing discount codes on the client. Let's say that the music store in our previous example wanted to reward its best customers by offering them a 50-percent-off discount. The music store emails these customers a special code that they can enter on the order form to receive the discount. In order to improve response time and save processing power on the Web server, the programmers implemented the discount logic in the client-side code rather than the server-side code.

```
<script type="text/javascript">

function processDiscountCode(discountCode) {
    if (discountCode == "HALF-OFF-MUSIC") {
        // redirect request to the secret discount order page
        window.location = "SecretDiscountOrderForm.html";
    }
}
</script>
```

The programmers must not have been expecting anyone to view the page source of the order form, because if they had, they would have realized that their “secret” discount code is plainly visible for anyone to find. Now everyone can have their music for half price.

In some cases, the sensitive string doesn’t even have to be a string. Some numeric values should be kept just as secret as connection strings or login credentials. Most e-commerce Web sites would not want a user to know the profit the company is making on each item in the catalog. Most companies would not want their employees’ salaries published in the employee directory on the company intranet.

It is dangerous to hardcode sensitive information even into server-side code, but in client-side code it is absolutely fatal. With just five seconds worth of effort, even the most unskilled “n00b” hacker can learn enough information to gain unauthorized access to sensitive areas and resources of your application. The ease with which this vulnerability can be exploited really highlights it as a critical danger. It is possible to extract hardcoded values from desktop applications using disassembly tools like IDA Pro or .NET Reflector, or by attaching a debugger and stepping through the compiled code. But this approach requires at least a modest level of time and ability, and again it only works for desktop applications. There is no guaranteed way to be able to extract data from server-side Web application code; this is usually only possible through some other configuration error such as an overly detailed error message or a publicly accessible backup file. With client-side JavaScript, though, all the attacker needs to do is click the “View Source” option in his Web browser. From a hacker’s point of view, this is as easy as it gets.

Comments and Documentation Included in Client-Side Code

The dangers of using code comments in client code have already been discussed briefly in Chapter 5, but it is worthwhile to mention them again here in the context of code transparency. Any code comments or documentation added to client side code will be accessible by the end user, just like the rest of the source code. When a programmer explains the logic of a particularly complicated function in source documentation, he is not only making it easier for his colleagues to understand, but also his attackers.

In general, any practice that increases code transparency should be minimized. On the other hand, it is important for programmers to document their code so that other people can maintain and extend it. The best solution is to allow (or force?) programmers to document their code appropriately during development, but not to deploy this code. Instead, a copy should be made with the documentation comments stripped out. This comment-less version of the code should be deployed to the production Web server. This approach is similar to the best practice concerning debug code. It is unreasonable and unproductive to prohibit programmers from creating debug versions of their applications, but these versions should never be deployed to a production environment. Instead, a mirrored version of the application minus the debug information is created for deployment. This is the perfect approach to follow for client-side code documentation as well.

This approach does require vigilance from the developers. They must remember to never directly modify the production code, and to always create the comment-less copy before deploying the application. This may seem like a fragile process that is prone to human error. To a certain extent this is true, but we are caught between the rock of security vulnerabilities (documented code being visible to attackers) and the hard place of un-maintainable code (no documentation whatsoever). A good way to mitigate this risk is to write a tool (or purchase one from a third party) that automatically strips out code comments. Run this tool as part of your deployment process so that it is not forgotten.

Security Note

Include comments and documentation in client-side code just as with server-side code, but never deploy this code. Instead, always create a comment-less mirrored version of the code to deploy.

Data Transformation Performed on the Client

Virtually every Web application has to handle the issue of transforming raw data into HTML. Any data retrieved from a database, XML document, binary file, or any other storage location must first be formatted into a human-readable structure before being displayed to a user. In traditional Web applications, this transformation is performed on the server, along with all the other HTML generation. However, Ajax applications are often designed in such a way that this data transformation is performed on the client instead of the server.

In some Ajax applications, the responses received from the partial update requests contain HTML ready to be inserted into the page DOM, and the client is not required to perform any data processing. Applications that use the ASP.NET AJAX `UpdatePanel` control work this way. In the majority of cases, though, the responses from the partial updates contain raw data in XML or JSON format that needs to be transformed into HTML before being inserted into the page DOM. There are many good reasons to design an Ajax application to work in this manner. Data transformation is computationally expensive. If we can get the client to do some of the “heavy lifting” of the application logic, we could improve the overall performance and scalability of the application by reducing the stress on the server. The downside to this approach is that performing data transformation on the client can greatly increase the impact of any code injection vulnerabilities such as SQL injection and XPath injection.

Code injection attacks can be very tedious to perform. SQL injection attacks in particular are notoriously frustrating. One of the goals of a typical SQL injection attack is to “break out” of the table referenced by the query and retrieve data from other tables. For example, assume that a SQL query executed on the server is as follows:

```
SELECT * FROM [Customer] WHERE CustomerId = <user input>
```

An attacker will try to inject his own SQL into this query in order to select data from tables other than the Customer table, such as the OrderHistory table or the CreditCard table. The usual method to accomplish this is to inject a `UNION SELECT` clause into the query statement (the injected code is shown in italics):

```
SELECT * FROM [Customer] WHERE CustomerId = x;
UNION SELECT * FROM [CreditCard]
```

The problem with this is that the results of `UNION SELECT` clauses must have exactly the same number and type of columns as the results of the original SELECT statement. The command shown in the example above will fail unless the Customer and CreditCard tables have identical data schemas. `UNION SELECT` SQL injection attacks also rely heavily on verbose error messages being returned from the server. If the application developers have taken the proper precautions to prevent this, then the attacker is forced to attempt blind SQL injection attacks (covered in depth in Chapter 3) which are even more tedious than `UNION SELECT`s.

However, when the query results are transformed into HTML on the client instead of the server, neither of these slow, inefficient techniques is necessary. A simple appended `SELECT` clause is all that is required to extract all the data from the database. Consider our previous SQL query example:

```
SELECT * FROM [Customer] WHERE CustomerId = <user input>
```

If we pass a valid value like ‘gabriel’ for the CustomerId, the server will return an XML fragment that would then be parsed and inserted into the page DOM.

```
<data>
  <customer>
    <customerid>gabriel</customerid>
    <lastname>Krahulik</lastname>
    <firstname>Mike</firstname>
    <phone>707-555-2745</phone>
  </customer>
</data>
```

Now, let’s try to SQL inject the database to retrieve the CreditCard table data simply by injecting a `SELECT` clause (the injected code is shown in italics).

```
SELECT * FROM [Customer] WHERE CustomerId = x;
SELECT * FROM [CreditCard]
```

If the results of this query are directly serialized and returned to the client, it is likely that the results will contain the data from the injected `SELECT` clause.

```
<data>
  <creditcard>
    <lastname>Holkins</lastname>
    <firstname>Jerry</firstname>
    <ccnumber>1234567812345678</ccnumber>
    <expirationDate>09-07-2010</expirationDate>
  </creditcard>
  <creditcard>
  …
</data>
```

At this point, the client-side logic that displays the returned data may fail since the data is not in the expected format. But this is irrelevant because the attacker has already won. Even if the stolen data is not displayed in the page, it was included with the server’s response, and any competent hacker will be using a local proxy or packet sniffing tool so that he can examine the raw contents of the HTTP messages being exchanged.

Using this simplified SQL injection technique, an attacker can extract out the entire contents of the backend database with just a few simple requests. A hack that before would require thousands of requests over a matter of hours or days might now take only a few seconds. This not only makes the hacker’s job easier, it also improves his chances of success since there is less likelihood that he will be caught by an intrusion detection system. Making 20 requests to the system is much less suspicious than making 20,000 requests to the system.

This simplified code injection technique is by no means limited to only SQL injection. If the server code is using an XPath query to retrieve data from an XML document, it may be possible for an attacker to inject his own malicious XPath clause into the query. Consider the following XPath query:

```
/Customer[CustomerId = <user input>]
```

An attacker could XPath inject this query as follows (the injected code is shown in italics):

```
/Customer[CustomerId = x] | /*
```

The | character is the equivalent of a SQL `JOIN` statement in XPath, and the `/*` clause instructs the query to return all of the data in the root node of the XML document tree. The data returned from this query will be all customers with a customer ID of “x” (probably an empty list) combined with the complete document. With a single request, the attacker has stolen the complete contents of the backend XML.

While the injectable query code (whether SQL or XPath) is the main culprit in this vulnerability, the fact that the raw query results are being returned to the client is definitely a contributing factor. This design antipattern is typically only found in Ajax applications and occasionally in Web services. The reason for this is that Web applications (Ajax or otherwise) are rarely intended to display the results of arbitrary user queries.

Queries are usually meant to return a specific, predetermined set of data to be displayed or acted on. In our earlier example, the SQL query was intended to return the ID, first name, last name, and phone number of the given customer. In traditional Web applications, these values are typically retrieved by element or column name from the query result set and written into the page HTML. Any attempt to inject a simplified “`;SELECT`” attack clause into a traditional Web application query may succeed; but since the raw results are never returned to the client and the server simply discards any unexpected values, there is no way for the attacker to exploit the vulnerability. This is illustrated in Figure 6-12.

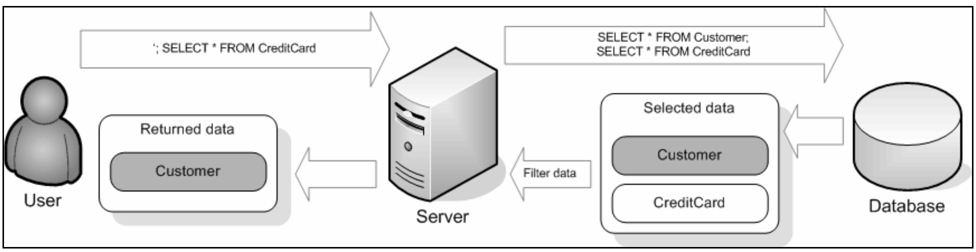


Figure 6-12
A traditional Web application using server-side data transformation will not return the attacker’s desired data

Compare these results with the results of an injection attack against an Ajax application that performs client-side data transformation (as shown in Figure 6-13), and you will see that it is much easier for an attacker to extract data from the Ajax application.

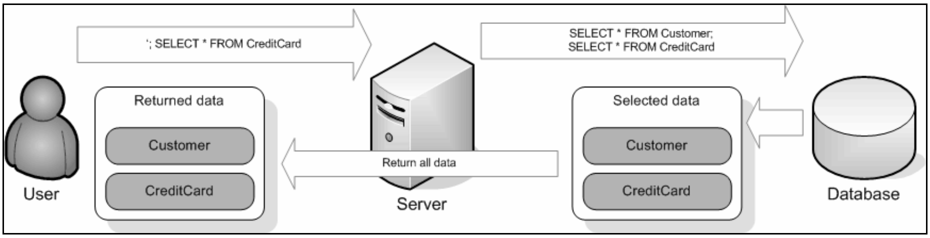


Figure 6-13
An Ajax application using client-side data transformation does return the attacker's desired data

Common implementation examples of this antipattern include:

- Use of the `FOR XML` clause in Microsoft SQL Server
- Returning .NET `System.Data.DataSet` objects to the client
- Addressing query result elements by numeric index rather than name
- Returning raw XPath/XQuery results

The solution to this problem is to implement a query output validation routine. Just as we validate all input to the query to ensure that it matches a predetermined format, we should also validate all output from the query to ensure that only the desired data elements are being returned to the client.

It is important to note that the choice of XML as the message format is irrelevant to the vulnerability. Whether we choose XML, JSON, comma-separated values, or any other format to send data to the client, the vulnerability can still be exploited unless we validate both the incoming query parameters and the outgoing results.

Security through Obscurity

Admittedly, the root problem in all of the specific design and implementation mistakes we've mentioned is not the increased transparency caused by Ajax. In MyLocalWeatherForecast.com, the real problem was the lack of proper authorization on the server. The programmers assumed that since the only pages calling the administrative functions already required authorization, then no further authorization was necessary. If they had implemented additional authorization checking in the server code, then the attacks would not have been successful. But while the transparency of the client code did not cause the vulnerability, it did contribute to the vulnerability by advertising the existence of the functionality. Similarly, it does an attacker little good to learn the data types of the server API method parameters if those parameters are properly validated on the server. But the increased transparency makes it more likely that any mistakes in the validation code will be found and exploited.

It may sound like we're advocating an approach of "security through obscurity", but in fact this is the complete opposite of the truth. It is generally a poor idea to assume that if your application is difficult to understand or reverse-engineer, then it will be safe from attack. The biggest problem with this approach is that it relies on the attacker's lack of persistence in carrying out an attack. There is no roadblock that obscurity can throw up against an attacker that cannot be overcome with enough time and patience. Some roadblocks are bigger than others; for example, 2048-bit asymmetric key encryption is going to present quite a challenge to a would-be hacker. But again, with enough time and patience (and cleverness) these problems are not insurmountable. The attacker may decide that the payout is worth the effort, or he may just see the defense as a challenge and tackle the problem that much harder.

That being said, while it's a bad idea to rely on security through obscurity, a little extra obscurity never hurts. Obscuring application logic raises the bar for an attacker, possibly stopping those without the skills or the patience to de-obfuscate the code. It is best to look at obscurity as one component of a complete defense and not a defense in and of itself. Banks don't advertise the routes and schedules that their armored cars take, but this secrecy is not the only thing keeping the burglars out: the banks also have steel vaults and armed guards to protect the money. Take this approach to securing your Ajax applications. Some advertisement of the application logic is necessary due to the requirements of Ajax, but always attempt to minimize it, and keep some (virtual) vaults and guards around in case someone figures it out.

Obfuscation

Code obfuscation is a good example of the tactic of obscuring application logic. Obfuscation is a method of modifying source code in such a way that it executes exactly the same but is much less readable to a human user.

JavaScript code can’t be encrypted since the browser wouldn’t know how to interpret it. The best that can be done to protect client-side script code is to obfuscate it. For example,

```
alert(“Welcome to JavaScript!”);
```

might be changed to this:

```
a = "lcome to J";
b = "al";
c = "avaScript!\")";
d = "ert(\"We";
eval(b + d + a + c);
```

These two blocks of JavaScript are functionally identical, but the second one is much more difficult to read. Substituting some Unicode escape characters into the string values makes it even harder:

```
a = "\u006c\u0063\u006fme t\u006f J";
b = "\u0061\u006c";
c = "\u0061v\u0061Sc\u0072ipt\u0021\")";
d = "e\u0072t(\"We";
eval(b + d + a + c);
```

There are practically an endless number of techniques that can be used to obfuscate JavaScript, several of which are described in the “Validating JavaScript Source Code” section of Chapter Four. In addition, there are some commercial tools available that will automate the obfuscation process and make the final code much more difficult to read than the samples given here. HTML Guardian™ by ProtWare is a good example. It’s always a good idea to obfuscate sensitive code, but keep in mind that obfuscation is not the same as encryption. An attacker will be able to reverse engineer the original source code given enough time and determination. Obfuscating code is a lot like tearing up a bank statement

– it doesn’t make the statement impossible to read, it just makes it harder by requiring the reader to reassemble it first.

Security Recommendation

Don’t:

Don’t confuse obfuscation with encryption. If an attacker really wants to read your obfuscated code, he will.

Do:

Do obfuscate important application logic code. Often this simple step is enough to deter the “script kiddie” or casual hacker who doesn’t have the patience or the skills necessary to recreate the original. However, always remember that *everything* that is sent to the client, even obfuscated code, is readable.

Conclusions

In terms of security, the increased transparency of Ajax applications is probably the most significant difference between Ajax and traditional Web applications. Much of traditional Web application security relies on two properties of server-side code; namely, that users can't see it, and that users can't change it. Neither of these properties hold true for client-side Ajax code. Any code downloaded to a user's machine can be viewed by the user. The application programmer can make this task more difficult, but in the end, a dedicated attacker will always be able to read and analyze the script executing on his machine. Furthermore, he can also change the script to alter the control flow of the application. Prices can be changed, authentication can be bypassed, and administrative functions can be called by unauthorized users. The solution is to keep as much business logic as possible on the server. Only server-side code is under the control of the developers – client-side code is under the control of attackers.

Rough Cuts

GET BEHIND THE SCENES

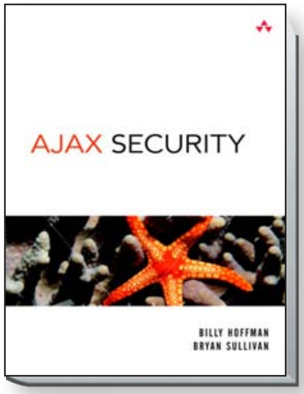
Start reading *Ajax Security* today

When you need to gain early access to information on cutting-edge technologies, turn to the ROUGH CUTS service from Safari Books Online.

Our publishers have granted a sneak peek into their authoring process. With ROUGH CUTS you'll access the book's content as it is being written.

You can choose to purchase online access to the book with unlimited viewing and PDF downloads of each revision, pre-purchase the print book, or get the best of both worlds – online access immediately and the print book later.

Any way you cut it, you will receive the finished product as soon as it is published, whether as a pdf or as a printed book.



ISBN: 0-321-49193-9

The hands-on, practical primer for professionals who want to prevent Ajax-related security vulnerabilities

Ajax has the potential to revolutionize the Internet as we know it, but with this potential comes equally dangerous new security threats. ***Ajax Security*** exposes the minefield of security vulnerabilities inherent in the Ajax framework and provides a guide for developers to safely navigate through it and create a secure application.

Each chapter begins with a myth about Ajax security which is then debunked. Throughout the book, you will find case studies of actual exploited Ajax vulnerabilities to illustrate key points. The authors also include specific recommendations for securing Ajax applications for each of the major Web programming languages (.NET, Java, and PHP) as well as for the popular new language, Ruby on Rails.

With this book, you will learn to

- Become familiar with security issue in the Web 2.0 world
- Create secure Mashup websites
- Identify vulnerabilities that may have crept in your code
- Keep new vulnerabilities out of your code

Geared towards software developers who want to create Ajax Web applications that are secure from attack, ***Ajax Security*** will also benefit QA testers and pen-testers who want to find vulnerabilities in the Ajax applications that they test.

Safari Books Online

Addison Wesley

PURCHASE THIS ROUGH CUT AT
http://safari.awprofessional.com/0321524403