Premature Ajax-ulation and You:

How to Recognize It, How to Avoid It, How to Treat it

Presented by:
Dr. Bryan Sullivan, MD
Dr. Billy Hoffman, MD*

*No, we're not really doctors.

Q: What is Premature Ajax-ulation (PA), and how does it affect me?

A: PA is caused by creating an Ajax application, or converting an existing application to use Ajax, without taking proper security precautions. PA can expose your web site to various health threats, such as:

- Embarrassing data leakage
- Unsightly privacy discharge
- Irritation in your compliance areas, such as SOX or PCI
- Unsanctioned backdoor access

Q: I think I might have PA. What are the symptoms?

A: There are a number of symptoms which indicate that you may have rushed into Ajax prematurely. These symptoms include:

Control flow manipulation of now-public operations.

In order to make partial page updates more useful, it can be worthwhile to increase the granularity of your server-side functions. It would be pointless to expose a single, lengthy "Do-It" operation that provides no user feedback while it is processing. Providing a more finely-grained server API also helps third party websites to create effective mashups from your application. On the other hand, **attackers can easily subvert the intended application workflow** and call functions out of order, change the parameter values of the calls, or omit the calls entirely.

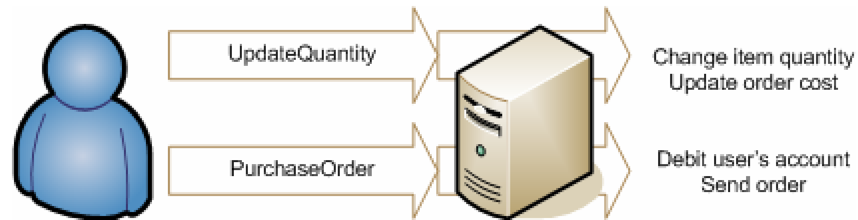
A real-world example of this design flaw is this password update function found in an actual ASP.NET AJAX application:

```
[ScriptMethod]
public string updatePassword(string custID, string newPassword)
{
  // connect to SQL Server
  SqlConnection conn = new SqlConnection(strCn);

  // create query
  string q = "UPDATE tblUser SET Password = '" +
    newPassword + "' WHERE CustomerID = '" + custID + "'";

  SqlCommand cmd = new SqlCommand(q, conn);
  cmd.ExecuteNonQuery();

  return "Password Updated to " + newPassword;
}
```

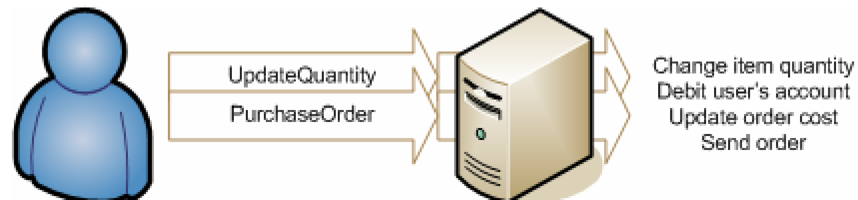
There are no authentication or authorization checks enforced on this code; anyone can call this function and change any user's password to any desired value. This code was probably originally a private method called during a page postback, and auth was enforced earlier in the call stack. However, once the application was converted to Ajax and the method visibility made public, the code became vulnerable.

Race conditions introduced into the application.

One of the great benefits of Ajax is being able to process data asynchronously, but **asynchronous operation implies multi-threaded operation**. The use of multiple threads of execution opens the door to classic threading problems such as race conditions and deadlocks.



For example, consider this e-commerce application that exposes server-side methods for updating item quantities and purchasing an order. If an attacker intentionally calls these methods back-to-back, without waiting to receive the response, it may be possible to subvert the intended logic:



Secrets inadvertently pushed into client-side code.

All code that executes on the client tier can be easily viewed and analyzed by an attacker. Any secrets stored in JavaScript, whether secret data like discount codes or database connection strings, or secret functionality like backdoor administrative access, will be found and exploited. This is a far easier mistake to make in an Ajax application than in a traditional web application because the client plays a larger role in data processing, presentation, and possibly storage. Even if your application doesn't explicitly store sensitive data in client-side storage, if you are caching arbitrary data from the server (such as record sets or large text blobs) you can inadvertently store sensitive data persistently on the client.

Do not rely on JavaScript obfuscation or string encryption to protect your secrets.

Functionality needlessly disclosed through script proxies.

Client-side code needs to be able to call server-side code, and so a certain amount of functionality needs to be disclosed. However, **disclosing unused server functions is dangerous**. You are needlessly increasing the attack surface of your application as well as providing a potential attacker with additional insight into the application's architecture.

Greatly increased susceptibility to injection attacks.

Performing data transformation such as XSLT functions on the client can improve application performance and scalability, but doing so also increases the risk from injection attacks. Vulnerabilities such as **SQL injection and XPath injection are much easier to exploit when data is transformed or filtered on the client**.

SQL injection vulnerabilities are notoriously frustrating for attackers to exploit manually, since any injected SQL "UNION SELECT" clause must contain the exact same number and types of parameter arguments as the original. Blind SQL injection attacks are even more tedious, potentially requiring an attacker to send thousands of requests (Is the first character of the first column of the first table an "a"? Is the first character of the first column of the first table a "b"?) However, if the query results are being returned directly to the client, as is common in Ajax partial page refreshes, these attacks can be avoided in place of simple "; SELECT" attacks.

Consider the following SQL query:

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

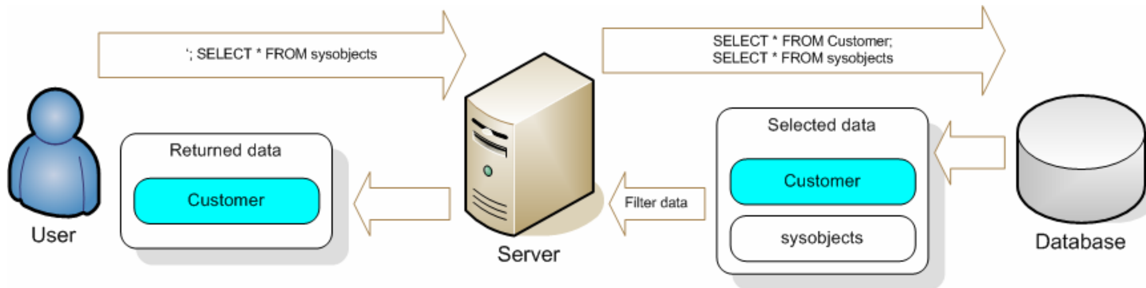
Passing a valid value like 'epresley' might cause the server to send back an XML fragment that would then be XSLT'd into the page DOM.

```
<data>
  <customer>
    <name>Elvis</name>
    <phone>555-555-5555</phone>
    <address>elvis@graceland.org</address>
  </customer>
</data>
```

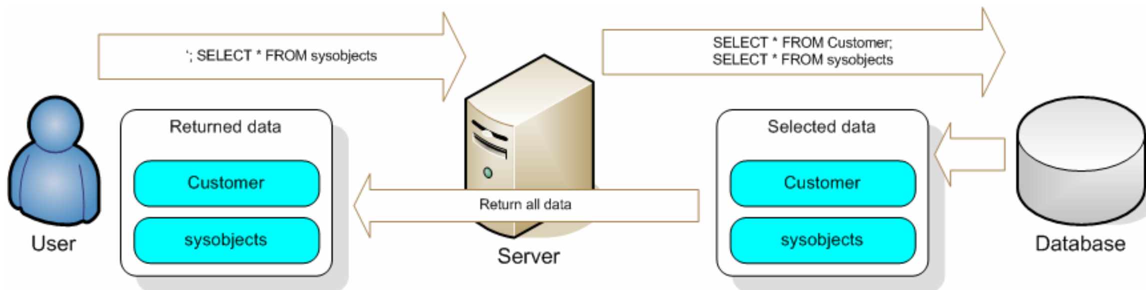
Now, consider what would happen if an attacker passed a SQL injection value to the server.

```
SELECT * FROM [Customer] WHERE id = 'x'; SELECT * FROM [sysobjects]
WHERE '1' = '1'
```

If the result set is filtered (or "sifted") by field name on the server, this attack would be fruitless. The sensitive data desired by the attacker may be returned from the query, but would not be sent to the client.



However, when the query results are returned as-is to the client for a partial update (aka client-side data binding), the attack is dramatically more successful:



```
<data>
  <customer/>
  <sysobjects>
    <name>sysrowsetcolumns</name>
    <id>4</id>
    <xtype>S</xtype>
    …
  </sysobjects>
  <sysobjects>
  …
</data>
```

At this point, the client-side transformation code may fail or display an empty page, but the damage has already been done since any smart attacker will be listening to the raw HTTP requests and responses with a proxy tool.

It is also important to note that it is not the use of XML as a data transport mechanism that is faulty. The same vulnerability would be present with JSON or any other custom format. The real source of the problem is the SQL injectable query, but the fact that the resulting data is being blindly sent to the client makes the injection much easier to exploit.

Q: What can I do to treat my PA?

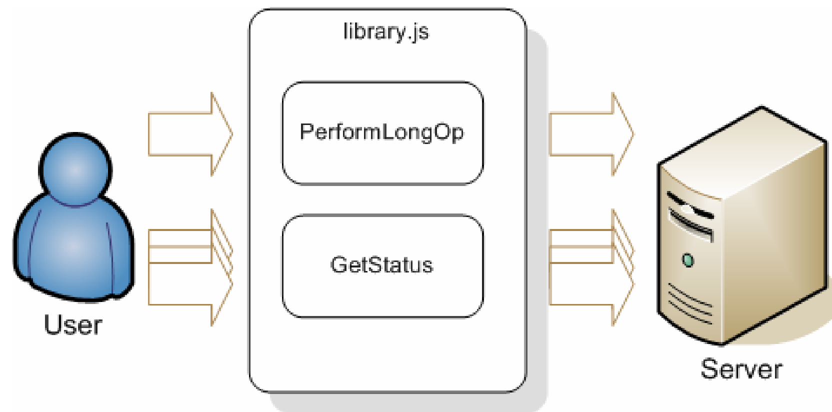
A: We recommend a multi-phase treatment process as follows:

Do not rely on code to be called the way you intended.

Unlike server-side code, client-side code is not a set of commands that the user must obey. Instead, **think of JavaScript code as a set of suggestions that the user may or may not choose to follow**. Every method exposed on the server can be called outside of its originally intended context. It is important to ensure that every method follows proper authentication and authorization procedures. For example, consider this pseudocode for trading stocks:

```
If User-Is-Authorized And User-Has-Sufficient-Funds Then Purchase-Stock
```

If this code is executed on the server, as a single atomic method, it should be secure. However, if this code is executed on the client, and the server exposes `User-Is-Authorized`, `User-Has-Sufficient-Funds`, and `Purchase-Stock` as three separate public methods, then it will likely be vulnerable. An attacker with a JavaScript debugger can simply skip over the `User-Is-Authorized` and `User-Has-Sufficient-Funds` calls and execute the `Purchase-Stock` method directly. Prerequisite method calls must always be enforced to be included as part of the operative method. This would include not just authorization functionality, but also resource allocation: don't open a file or lock a mutex in one function and access it or free it in another.



If the intent of breaking a large, atomic method into smaller, more granular methods is simply to provide more feedback to the user, it may be safer to leave the large method as-is and expose a second `Get-Status` method that the client can poll asynchronously (as shown in the diagram above).

Beware of Client-side Storage and “offline” Ajax.

Storing secrets in client-side code is the digital equivalent of hiding your door key under the welcome mat. Everyone looks there! **All secrets should be present only in the server code**. This includes test credentials, connection strings, pricing logic (in fact, most business logic), and even code comments.

This prescription against storing secrets also extends to potentially sensitive data entered by the user. For example, an Ajax-based word processor should not store users’ documents in any kind of client local data store. Long term storage of documents (or even temporary caching of parts of documents) in persistent client-side storage mechanisms is dangerous. These storage mechanisms include HTTP cookies, Flash Local Shared Objects, Mozilla’s DOM storage, and Internet Explorer’s userData. Storage mechanisms like these can be vulnerable to shared storage attacks, such as:

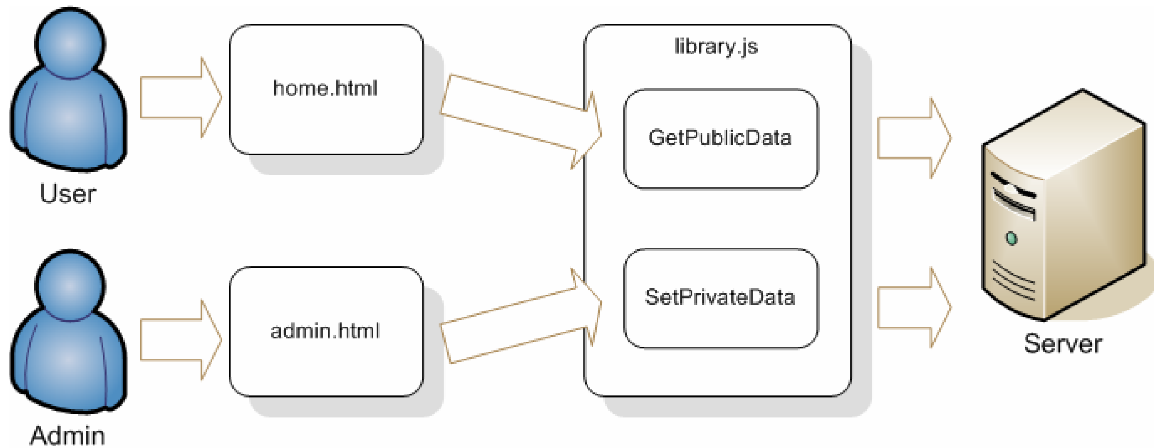
- Cross domain attacks. Improperly restricting which domains can access the stored data will allow malicious sites to read and modify the data. For example, allowing all .com domains is grossly overly permissive.
- Cross directory attacks. Similar to cross domain attacks, cross directory attacks are possible when developers improperly restrict which directories can access the stored data. Common victims are social networking sites where all users are assigned folders on the same web server. For example, every MySpace user is given personal space to create their page at http://www.myspace.com/<username>. If directory restrictions are not set properly, you could inadvertently grant lonelygirl15 complete access to your personal data.
- Cross port attacks. Again, similar to cross domain and cross directory attacks, cross port attacks occur when a web server outside your control is running on the same host name as your server, but on a different port. This is obviously a fairly rare situation, but when it does occur it is exceptionally dangerous since all access restrictions will be bypassed.

Extra care must be taken when developing “offline Ajax” applications that allow the user to continue working even when disconnected from the Internet using frameworks such as Google Gears. These frameworks provide local SQL databases and worker threads. These features enable developers to implement more server functionality on the client to increase the productivity of the application while in “offline” mode. All this additional logic and data provides an attacker

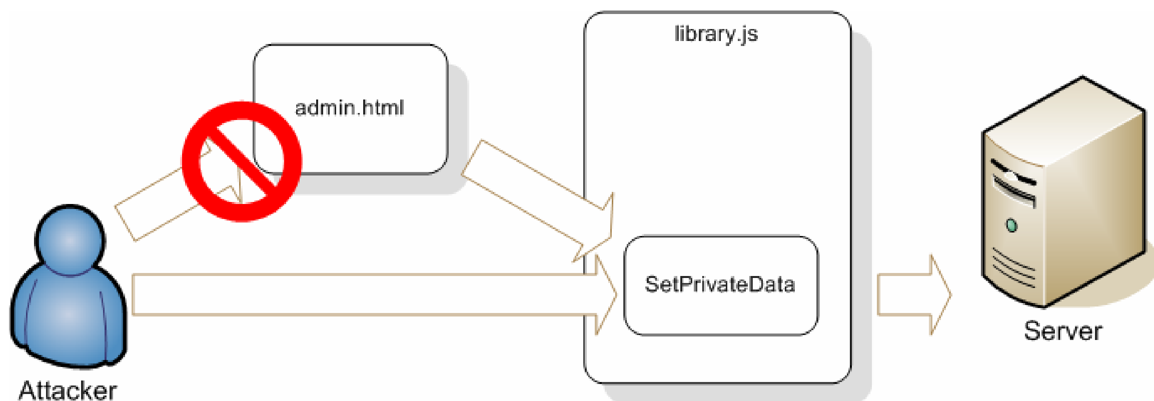
with more opportunities to reverse engineer the application and also provides insight into the server-side code and database schemas.

Only expose required server-side methods.

When you publicly expose server methods for your Ajax applications, you are essentially creating an API for anyone to call. This includes methods exposed through use of a third-party Ajax framework. Any **unused framework methods should be removed or made inaccessible**, if possible. This will reduce your attack surface without impacting functionality.



This recommendation holds not just for third-party framework APIs, but also ones that you develop yourself. Server methods that should only be available to highly privileged users (such as site administration methods) ideally should not be visible to less privileged users. In the diagram above, both the generally visible `GetPublicData` method and the administrative `SetPrivateData` method are defined within the same script proxy library file. While the `SetPrivateData` method is only referenced from the administrative page admin.html, it is plainly visible to any attacker who views the script file. If authentication and/or authorization are enforced only on the page and not on the call, an attacker could bypass the intended auth mechanisms.



In this case, the fault is not the fact that an attacker could discover the `SetPrivateData` method, but rather that the method was not properly secured. However, the unnecessary disclosure of the method to unprivileged users did make the application more susceptible to attack. Hiding the administrative methods from the general public would have lessened the chance that the authorization vulnerability would ever have been found.

It may sound like we are advocating security through obscurity. This is not true. We are simply advocating augmenting proper security with as much additional obscurity as possible.

Validate all user input on the server.

Never trust the user! **All user input should be validated for correct format and length**, and any input that does not pass these validation checks should not be processed (aka whitelist validation). If a method parameter is supposed to contain a phone number, make sure it matches the pattern of a phone number. Regular expressions are perfect for this. It is critically important to perform this validation on the server, and not just the client, since client-side JavaScript can be trivially circumvented.

The Regular Expression Library site (http://regexlib.com/) is a good place to go for regular expressions. There is little point in "reinventing the wheel" for every regex you need, especially for tricky ones like email addresses.

Security-conscious developers have known for years about the need to validate form input values, but in an Ajax solution it is just as critical to validate the server-side method parameter input values. Not only are these inputs easy to overlook since they have no visible representation in the user interface, but they also may expose vulnerabilities that cannot be exploited through a web browser. For example, the client-side code executing in the browser may enforce the rule that only numeric values can be passed for a method argument, or that only a valid US state abbreviation such as "CA" or "NV" can be passed. But, through the use of a raw HTTP request tool such as telnet or netcat, it is trivial for an attacker to send any value he wants.

"Sift" data by name on the server.

Do not return query results from a SQL database, XML document, or any other datastore directly to the client without first "sifting" them. Just as you ensure that all input to the query matches the intended format, you must also **ensure that all output from the query matches the intended format**.

For example, say we want to retrieve the first and last names of a customer and return them to the client for display:

```
SELECT FirstName, LastName FROM Customer WHERE CustomerID = <input>
```

If the server code simply collects the results of this query and returns them to the client, it could be vulnerable:

```
// this is bad, vulnerable code
SqlDataReader reader = selectCommand.ExecuteReader();
do
{
  while (reader.Read())
  {
    for (int i = 0; i < reader.FieldCount; i++)
      output += reader.GetValue(i).ToString() + ",";
  }
}
while (reader.NextResult());
```

A common practice in ASP.NET is to return a .NET DataSet object to the client. This is also dangerous, for the same reason.

```
// this is bad, vulnerable code
SqlDataAdapter adapter = new SqlDataAdapter(selectCommand, connection);
DataSet resultSet = new DataSet();
adapter.Fill(resultSet);
return resultSet;
```

A more secure technique is to pull the intended fields from the result set *by name* and return only those fields.

```
// this is more secure code
SqlDataReader reader = selectCommand.ExecuteReader();
do
{
  while (reader.Read())
  {
    output += reader["Customer.FirstName"].ToString();
    output += ",";
    output += reader["Customer.LastName"].ToString();
    output += ";";
  }
}
while (reader.NextResult());
```

Slow down (and think about security).

The trend toward Rapid Application Development (RAD) can be very BAD for security. For too many developers, it is sufficient just to know that their application works as intended. However, it is just as important to **understand why the application works the way it does**. Without understanding the "why", developers could be inadvertently placing business logic in the client-side code, or relying on server-side methods to be called in a specific order. This is especially relevant when using third-party frameworks.

Choose your partners carefully.

Ajax development is a new field and consistent, quality documentation is lacking from even the most popular Ajax frameworks. As a result, many novice developers directly implement sample code or ideas using "copy+paste+edit until it compiles" programming when they find an article describing how to do something. Little thought is given to who is supplying the advice.

Unfortunately, many of the most popular Ajax resources available to developers give poor or no advice on security. Many popular Ajax books discuss insecure coding practices and contain code samples vulnerable to threats like remote file disclosure, XSS, and SQL injection. Popular message boards, news sites, and forums for Ajax development also rarely mention security considerations, if they are even mentioned at all. Some even offer blatantly false security recommendations, such as converting Ajax callback functions from GET to POST to defend against Cross Site Request Forgery attacks. Ask yourself: Are these people competent? How long have they been doing this? What types of security testing have been performed on the example?

When accepting sample code, programming advice, or design patterns from 3rd parties you should think strongly about where that information is coming from. Insecure programming practices are infectious. Remember what your mother said: Just because everyone else is doing it doesn't mean it's the right way to do it.

Consider abstinence.

Ajax applications are more difficult to design, develop, and test for security than traditional web applications. If your application cannot clearly benefit from the addition of Ajax, you should probably consider alternatives. In other words, **sometimes the safest way to do Ajax is not to do Ajax**. Some examples of applications that have no real need for Ajax include:

- Applications whose content is both relatively small and relatively static. This data could easily be cached in the web page when it is requested. For example, retrieving stock prices is a good use of Ajax since they change so frequently. Auto-complete is a good use of Ajax since it would be impractical to cache the entire dictionary. But retrieving weather forecast data would not be a good use of Ajax, since it is both fairly small (probably less than 1Kb) and generally doesn't change more often than once every few hours.
- Ajax pages that partially refresh 90% or more of their page content. If this much of the page needs to be refreshed after every request, it is probably safer to just use complete page updates.
- Applications that are asynchronous in name only. Unless the user can do something useful while waiting for the server to respond to his request, there is no reason to pay the price for the complexity of an asynchronous architecture.

Q: What are my chances of recovery?

A: If you choose to ignore these warnings, it may already be too late for you. If you do follow our advice and delay your Ajax gratification until all parties reach security satisfaction (preferably at the same time), the prognosis is good for a secure relationship between you and your users.

For more information, please contact us at:
http://www.spidynamics.com/spilabs
bryan.sullivan@spidynamics.com
billy.hoffman@spidynamics.com

Learn more about Premature Ajax-ulation in our upcoming book, *Ajax Security*, published by Addison-Wesley. *Ajax Security* will be available in early November 2007.