NoSQL, But Even Less Security

Bryan Sullivan, Senior Security Researcher, Adobe Secure Software Engineering Team
Agenda

Eventual Consistency
REST APIs and CSRF
NoSQL Injection
SSJS Injection
NoSQL databases
Choose any two:

- Availability
- Consistency
- Partition Tolerance
Eventual consistency in social networking

Pizza for lunch today

Kymberlee

Load balancer

NoSQL node 1

NoSQL node 2

NoSQL node 3
Writes don't propagate immediately
Reading stale data

I wonder what Kymberlee is having for lunch...

Adrian

Load balancer

NoSQL node 1

NoSQL node 2

NoSQL node 3
Reading stale data – a more serious case

I wonder what the price of AMZN is right now...

Bryan

Load balancer

NoSQL node 1
$200

NoSQL node 2
$200

NoSQL node 3
$199
Agenda

Eventual Consistency
REST APIs and CSRF
NoSQL Injection
SSJS Injection
Authentication is unsupported or discouraged

- From the MongoDB documentation
  - “One valid way to run the Mongo database is in a trusted environment, with no security and authentication”
  - This “is the default option and is recommended”

- From the Cassandra Wiki
  - “The default AllowAllAuthenticator approach is essentially pass-through”

- From CouchDB: The Definitive Guide
  - The “Admin Party”: Everyone can do everything by default

- Riak
  - No authentication or authorization support
Port scanning

- If an attacker finds an open port, he’s already won...

<table>
<thead>
<tr>
<th>Database</th>
<th>Default Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>27017</td>
</tr>
<tr>
<td></td>
<td>28017</td>
</tr>
<tr>
<td></td>
<td>27080</td>
</tr>
<tr>
<td>CouchDB</td>
<td>5984</td>
</tr>
<tr>
<td>Hbase</td>
<td>9000</td>
</tr>
<tr>
<td>Cassandra</td>
<td>9160</td>
</tr>
<tr>
<td>Neo4j</td>
<td>7474</td>
</tr>
<tr>
<td>Riak</td>
<td>8098</td>
</tr>
</tbody>
</table>
Port Scanning Demo
Port scanning

- If an attacker finds an open port, they've already won...

<table>
<thead>
<tr>
<th>Database</th>
<th>Port(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>27017, 2801, 27080</td>
</tr>
<tr>
<td>CouchDB</td>
<td>5984</td>
</tr>
<tr>
<td>Hbase</td>
<td>9000</td>
</tr>
<tr>
<td>Cassandra</td>
<td>9160</td>
</tr>
<tr>
<td>Neo4j</td>
<td>7474</td>
</tr>
<tr>
<td>Riak</td>
<td></td>
</tr>
</tbody>
</table>
REST document API examples (CouchDB)

- **Retrieve a document**
  
  GET /mydb/doc_id HTTP/1.0

- **Create a document**
  
  POST /mydb/ HTTP/1.0
  
  ```json
  {
    "album" : "Brothers",
    "artist" : "Black Keys"
  }
  ```

- **Update a document**
  
  PUT /mydb/doc_id HTTP/1.0
  
  ```json
  {
    "album" : "Brothers",
    "artist" : "The Black Keys"
  }
  ```

- **Delete a document**
  
  DELETE /mydb/doc_id?rev=12345 HTTP/1.0
Cross-Site Request Forgery (CSRF) firewall bypass

Attacker

Firewall

Victim

$dbserver:1234/delete_db$

$<img src="dbserver:1234/delete_db" />$

$dbserver:1234$
REST document API examples (CouchDB)

- **Retrieve a document**
  GET /mydb/doc_id HTTP/1.0

- **Create a document**
  POST /mydb/ HTTP/1.0
  {
   "album" : "Brothers",
   "artist" : "Black Keys"
  }

- **Update a document**
  PUT /mydb/doc_id HTTP/1.0
  {
   "album" : "Brothers",
   "artist" : "The Black Keys"
  }

- **Delete a document**
  DELETE /mydb/doc_id?
  rev=12345 HTTP/1.0
Traditional GET-based CSRF

- Easy to make a potential victim request this URL
- But it doesn’t do the attacker any good
- He needs to get the data back out to himself
RIA GET-based CSRF

```javascript
<script>
    var xhr = new XMLHttpRequest();
    xhr.open('get', 'http://nosql:5984/_all_dbs');
    xhr.send();
</script>

- Just as easy to make a potential victim request this URL
- Same-origin policy won’t allow this (usually)
- Same issue for PUT and DELETE
POST-based CSRF

```html
<form method=post action='http://nosql:5984/db'>
    <input type='hidden' name='{"data"}' value='"' />
</form>

<script>
    // auto-submit the form
</script>

- Ok by the same-origin policy!
REST-CSRF Demo
POST is all an attacker needs

Insert arbitrary data

Insert arbitrary script data

Execute any REST command from inside the firewall
Agenda

Eventual Consistency
REST APIs and CSRF
NoSQL Injection
SSJS Injection
Most developers believe they don’t have to worry about things like this

“…with MongoDB we are not building queries from strings, so traditional SQL injection attacks are not a problem.”

- MongoDB Developer FAQ

They’re mostly correct
MongoDB and PHP

- MongoDB expects input in JSON array format
  ```javascript
  find( { 'artist' : 'The Black Keys' } )
  ```

- In PHP, you do this with associative arrays
  ```php
  $collection->find(array('artist' => 'The Black Keys'));
  ```

- This makes injection attacks difficult
- Like parameterized queries for SQL
You also use associative arrays for query criteria

```
find( { 'album_year' : { '$gte' : 2011} } )
find( { 'artist' : { '$ne' : 'Lady Gaga' } } )
```

But PHP will automatically create associative arrays from querystring inputs with square brackets

```
page.php?param[foo]=bar
param == array('foo' => 'bar');
```
NoSQL Injection Demo
$where queries

- The $where clause lets you specify script to filter results

```javascript
find( { '$where' : 'function() { return artist == "Weezer"; }' } )

find ( '$where' : 'function() {
   var len = artist.length;
   for (int i=2; i<len; i++) {
      if (len % i == 0) return false;
   }
   return true; }
)
```
NoSQL Injection Demo #2
Agenda

Eventual Consistency
REST APIs and CSRF
NoSQL Injection
SSJS Injection
Browser wars have given us incredibly fast and powerful JS engines.

- V8
- WebKit Nitro
- SpiderMonkey Rhino

- Used for a lot more than just browsers
- Like NoSQL database engines...
Server-side JavaScript injection vs. XSS

- Client-side JavaScript injection (aka XSS) is #2 on OWASP Top Ten
  - Use it to steal authentication cookies
  - Impersonate victim
  - Create inline phishing sites
  - Self-replicating webworms ie Samy

- It’s really bad.
- But server-side is much worse.
Server-Side Javascript Injection (SSJI)
SSJI red flags

- $where clauses
  - Built with user input
  - Injected from querystring manipulation

- eval() clauses

- Map/Reduce

- Stored views/design docs
  - More CSRF possibilities here
Wrapping Up
Conclusions

1. Always use authentication/authorization.
   - Firewalls alone are not sufficient
   - Sometimes you may have to write your own auth code
   - This is unfortunate but better than the alternative

2. Be extremely careful with server-side script.
   - Validate, validate, validate
   - Escape input too
Read my blog: http://blogs.adobe.com/asset
Email me: brsulliv