No SQL! no injection?
A talk on the state of NoSQL security

IBM Cyber Security Center of Excellence
Aviv Ron
Alexandra Shulman-Peleg
Anton Puzanov
Aviv Ron

- Security Researcher for **IBM Cyber Security Center of Excellence** at Beer Sheva
  🐦 @aviv_ron
- Focus on Application Security in the cloud
- Ongoing research on new and emerging application vulnerabilities for IBM AppScan, Application Security Testing
- Joined IBM at 2014, prior with Intel for 9 years

dressed up as a security researcher

when I’m not a security researcher
10 years of SQL Injection

It has been 10 years since the publishing of the first SQL Injection research paper and yet, today we are seeing more devastating exploitation of the SQL Injection attack than ever before. Companies, data, and networks are being broken open through this simple attack that floats right through the firewall on port 80 or 443 and opens up the soft internal networks of many organizations.

14 Years of SQL Injection and still the most dangerous vulnerability

Category: Web Security - Tags: sql injection, web application security - Thu, 22 Aug 2013, by Alex Baker

Ever since the advent of the computer, there have always been people trying to hack them. William D. Matthew of MIT discovered, for the first time in 1965, the vulnerability in the ENIAC program that opened up the door to IBM-7094 in 1955.

The History of SQL Injection, the Hack That Will Never Go Away

Written by JOSEPH COX

20 November 2015 // 02:00 PM CET

One of the hackers suspected of being behind the TalkTalk breach, which led to the personal details of at least 150,000 people being stolen, used a vulnerability discovered two years before he was even born.
Who doesn’t know “Little Bobby Tables”

http://xkcd.com/327/
Fast forward to 2015
Not only SQL

<table>
<thead>
<tr>
<th>Rank</th>
<th>DBMS</th>
<th>Database Model</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Oracle</td>
<td>Relational DBMS</td>
<td>1446.13</td>
</tr>
<tr>
<td>2.</td>
<td>MySQL</td>
<td>Relational DBMS</td>
<td>1284.58</td>
</tr>
<tr>
<td>3.</td>
<td>Microsoft SQL Server</td>
<td>Relational DBMS</td>
<td>1149.11</td>
</tr>
<tr>
<td>4.</td>
<td>MongoDB +</td>
<td>Document store</td>
<td>278.59</td>
</tr>
<tr>
<td>5.</td>
<td>PostgreSQL</td>
<td>Relational DBMS</td>
<td>268.31</td>
</tr>
<tr>
<td>6.</td>
<td>DB2</td>
<td>Relational DBMS</td>
<td>197.65</td>
</tr>
<tr>
<td>7.</td>
<td>Microsoft Access</td>
<td>Relational DBMS</td>
<td>142.19</td>
</tr>
<tr>
<td>8.</td>
<td>Cassandra +</td>
<td>Wide column store</td>
<td>104.89</td>
</tr>
<tr>
<td>9.</td>
<td>SQLite</td>
<td>Relational DBMS</td>
<td>102.30</td>
</tr>
<tr>
<td>10.</td>
<td>Redis</td>
<td>Key-value store</td>
<td>94.55</td>
</tr>
</tbody>
</table>

According to [http://db-engines.com](http://db-engines.com)
Applications of NoSQL

BIG DATA

REAL TIME WEB

PERFORMANCE

FLEXIBILITY

SCALABILITY

Images are under Creative Commons license and are attributed to their creators
It’s not that relational databases are bad
We are just saying tables are not the solution for EVERYTHING
SO… no sql, no worries?
Not really…

Introducing NoSQL Injections
A look at mongodb

db.books.insert(
    {
        title: 'The Hobbit',
        author: 'J.R.R. Tolkien'
    }
)

db.books.find(
    {
        title: 'The Hobbit',
        author: 'J.R.R. Tolkien'
    }
)

array('title' => 'The hobbit', 'author' => 'J.R.R. Tolkien');
Login

Username: 
Password: 

HTTP POST

username=tolkien&password=hobbit

db->logins->find(array(
    "username"=>$_POST["username"],
    "password"=>$_POST["password"]));

{ username: 'tolkien', password: 'hobbit' }
Login

Username: [input]
Password: [input]

HTTP POST

username[$ne]=1 & password[$ne]=1

```php
db->logins->find(
  array("username"=>array("$ne" => 1),
    "password"=> array("$ne" => 1));
```

```json
{ username: { $ne: 1 }, password: { $ne: 1 } }
```
PHP Parameter pollution

db->logins->find(
    array("$where"=>"function() { return this.price < 100 }"));
From PHP documentation:

“Please make sure that for all special query operators (starting with $) you use single quotes so that PHP doesn't try to replace "$exists" with the value of the variable $exists.”
Not only in php
let’s take a look at JavaScript
Login

Username: 
Password: 

HTTP POST

username=tolkien&password=hobbit

string query =
{"username: " + post_username + ", password: " + post_password + "" }"

{ username: 'tolkien', password: 'hobbit' }
Login

Username:  
Password:  

HTTP POST

username=tolkien', $or: [ {}, { 'a': 'a', password=''} ], $comment: 'hacked'

string query =
"{ username: " + post_username + ", password: " + post_password + " }"

{ username: 'tolkien', $or: [ {}, { 'a': 'a', password: '' } ], $comment: 'hacked' }
NoSQL Javascript Injection
$map = "function() {
    for (var i = 0; i < this.items.length; i++) {
        emit(this.name, this.items[i].$param);
    }
};"

$reduce = "function(name, sum) {return Array.sum(sum); }";

$opt = "{ out: 'totals' }";

$db->execute("db.stores.mapReduce($map, $reduce, $opt);");
Attack on map reduce javascript

a);});function(kv) { return 1; }, { out: 'x'
});db.injection.insert({success:1});return
1;db.stores.mapReduce(function() { { emit(1,1
Attack on map reduce javascript

db.stores.mapReduce(function() {
  for (var i = 0; i < this.items.length; i++) {
    emit(this.name, this.items[i].a);
  }
}, function(kv) { return 1; }, { out: 'x' });

db.injection.insert({success:1});return 1;db.stores.mapReduce(function() {
  { emit(1,1

db.stores.mapReduce(function() {
  for (var i = 0; i < this.items.length; i++) {
    emit(this.name, this.items[i].a);
  }
}, function(kv) { return 1; }, { out: 'x' });

db.injection.insert({success:1});
return 1;db.stores.mapReduce(function() {
  { emit(1,1);

function(name, sum) { return Array.sum(sum); }, { out: 'totals' }});
Now – let’s Have some REST

Returns all documents (query and options can be sent in GET body)

GET /db/collection?query=%7B%22isDone%22%3Afalse%7D

Returns all documents satisfying query

GET /db/collection?query=%7B%3Atrue%7D

Ability to add options to GET

GET /db/collection/id

Returns document with id

POST /db/collection

Insert new document in collection

PUT /db/collection/id

Update document with id (updated document in PUT body)

DELETE /db/collection/id
CSRF attack on NoSQL REST API

1. Intranet (Secure Network) to Malicious website
2. Malicious website to Internet (Public Network)
3. Employee sends notification to set all items price to zero

Code snippet:
```html
<form method="post" action="http://nosql.internal.com/items/price">
  <input type="number" name="price" value="0" />
</form>
<script>
  document.forms[0].submit();
</script>
```
CSRF attack on NoSQL REST API

16.2.1. Description

Apache CouchDB versions prior to version 0.11.1 are vulnerable to Cross Site Request Forgery (CSRF) attacks.

16.2.2. Mitigation

All users should upgrade to CouchDB 0.11.2 or 1.0.1.

Upgrades from the 0.11.x and 0.10.x series should be seamless.

Users on earlier versions should consult with upgrade notes.

16.2.3. Example

A malicious website can POST arbitrary JavaScript code to well known CouchDB installation URLs (like http://localhost:5984/) and make the browser execute the injected JavaScript in the security context of CouchDB’s admin interface Futon.

Unrelated, but in addition the JSONP API has been turned off by default to avoid potential information leakage.
Beware of third party API’s

DrowsyDromedary
DrowsyDromedary is a REST layer for MongoDB based on Ruby.

Crest (Node.js)
Crest is a REST API Server for MongoDB built in node.js using the MongoDB Node Native driver.

AMID
AMID is a REST interface for MongoDB. Written in Node.js, supports multi-threading and a modular architecture to perform custom search operation. AMID also provides an optional extjs GUI for queries: AMIDGUI.

Mongodb Java REST server
Mongodb Java REST server based on Jetty.

Kule
Kule is a customizable REST interface for MongoDB based on Python.

DreamFactory
DreamFactory is an open source backend with a REST API for MongoDB. DreamFactory on GitHub.

HTTP Interfaces

Sleepy Mongoose (Python)
Sleepy Mongoose is a full featured HTTP interface for MongoDB.
Defending against risks
Defenses

- Injections
  - **Sanitize all user input** – do not assemble JSON from strings
  - If possible disable Javascript execution on DB else be careful when inserting user input to javascript
  - Beware of $ operators in PHP

- CSRF
  - Check your HTTP API framework for CSRF protection (NO JSONP, use of random token)

- General
  - Use automatic tools for application security testing that cover NoSQL vulnerabilities such as IBM AppScan
  - Use of role based access control and the principal of least privilege

NoSQL databases suffer from the same security issues their relational siblings do
Thank you for attending!