NoSQL, But Even Less Security

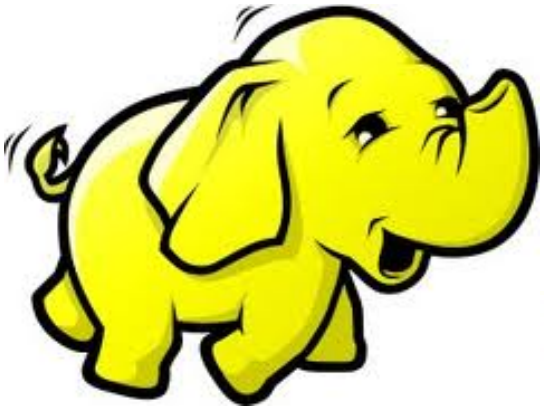
Bryan Sullivan, Senior Security Researcher, Adobe Secure Software Engineering Team

Agenda

Eventual Consistency
REST APIs and CSRF
NoSQL Injection
SSJS Injection

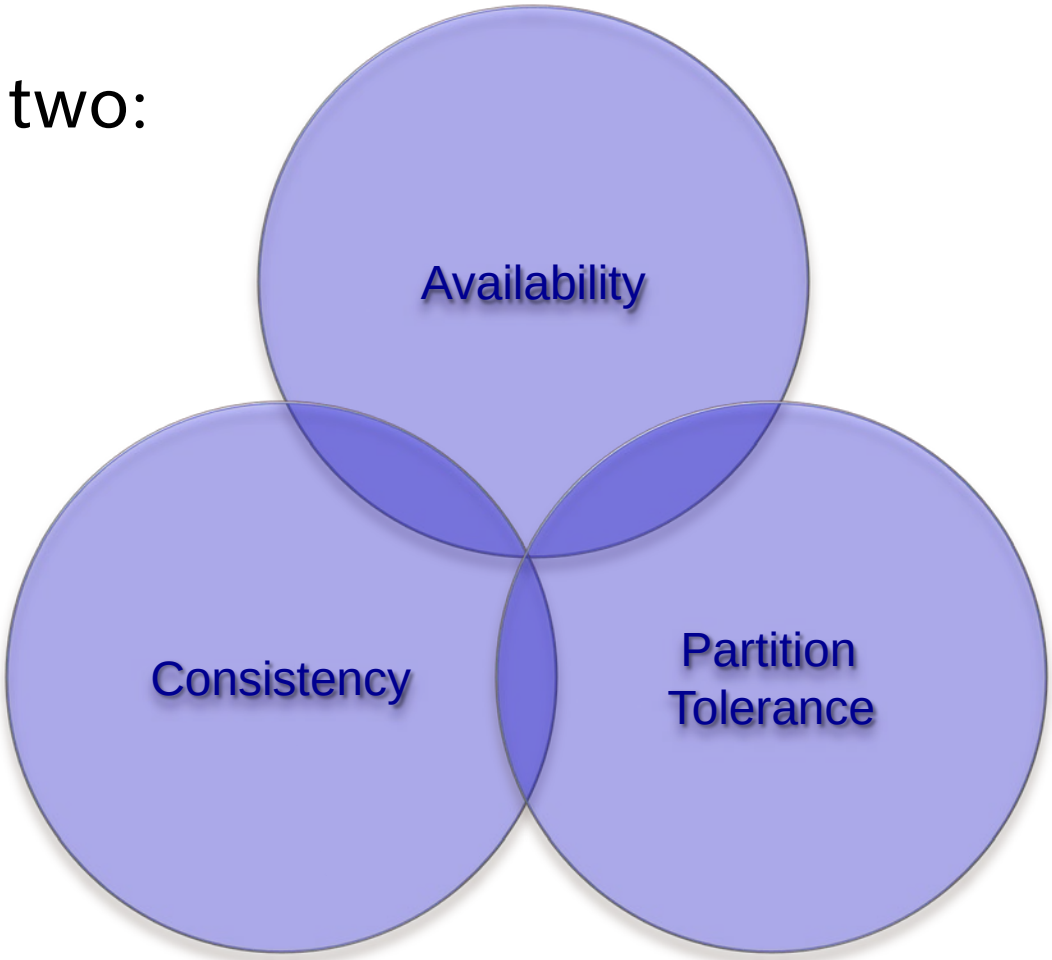
Adobe

NoSQL databases



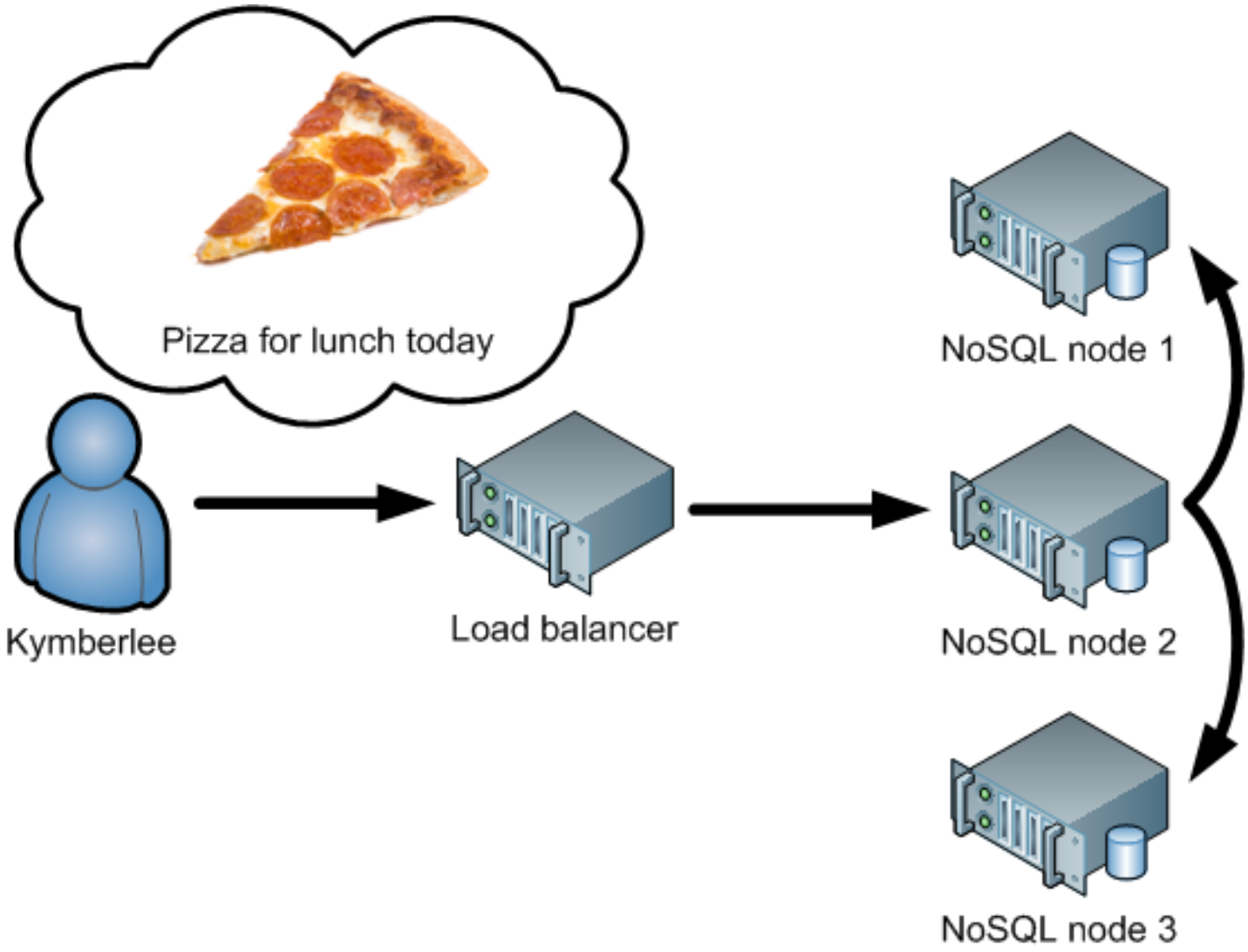
Eric Brewer's CAP Theorem

Choose any two:



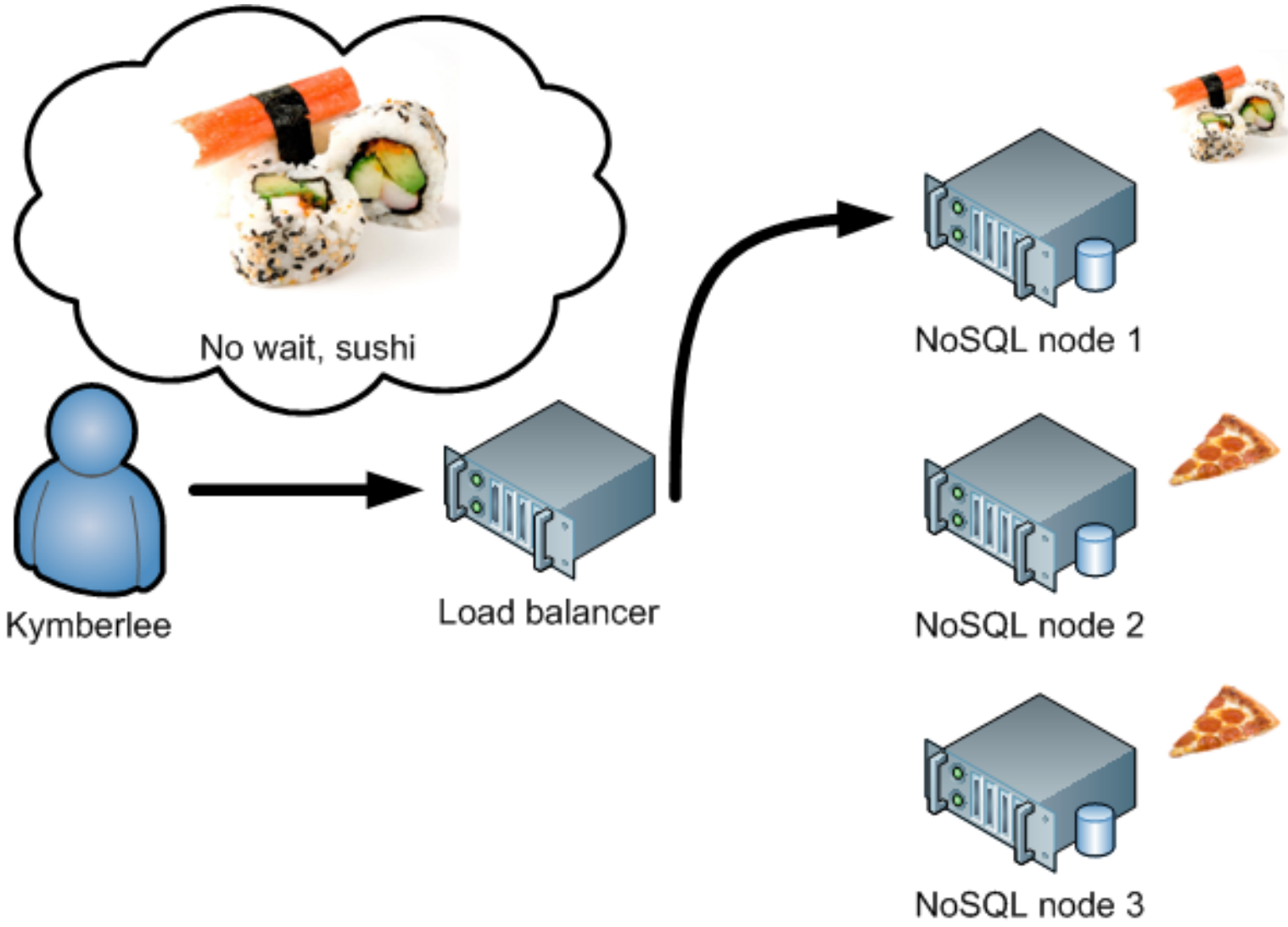
Adobe

Eventual consistency in social networking



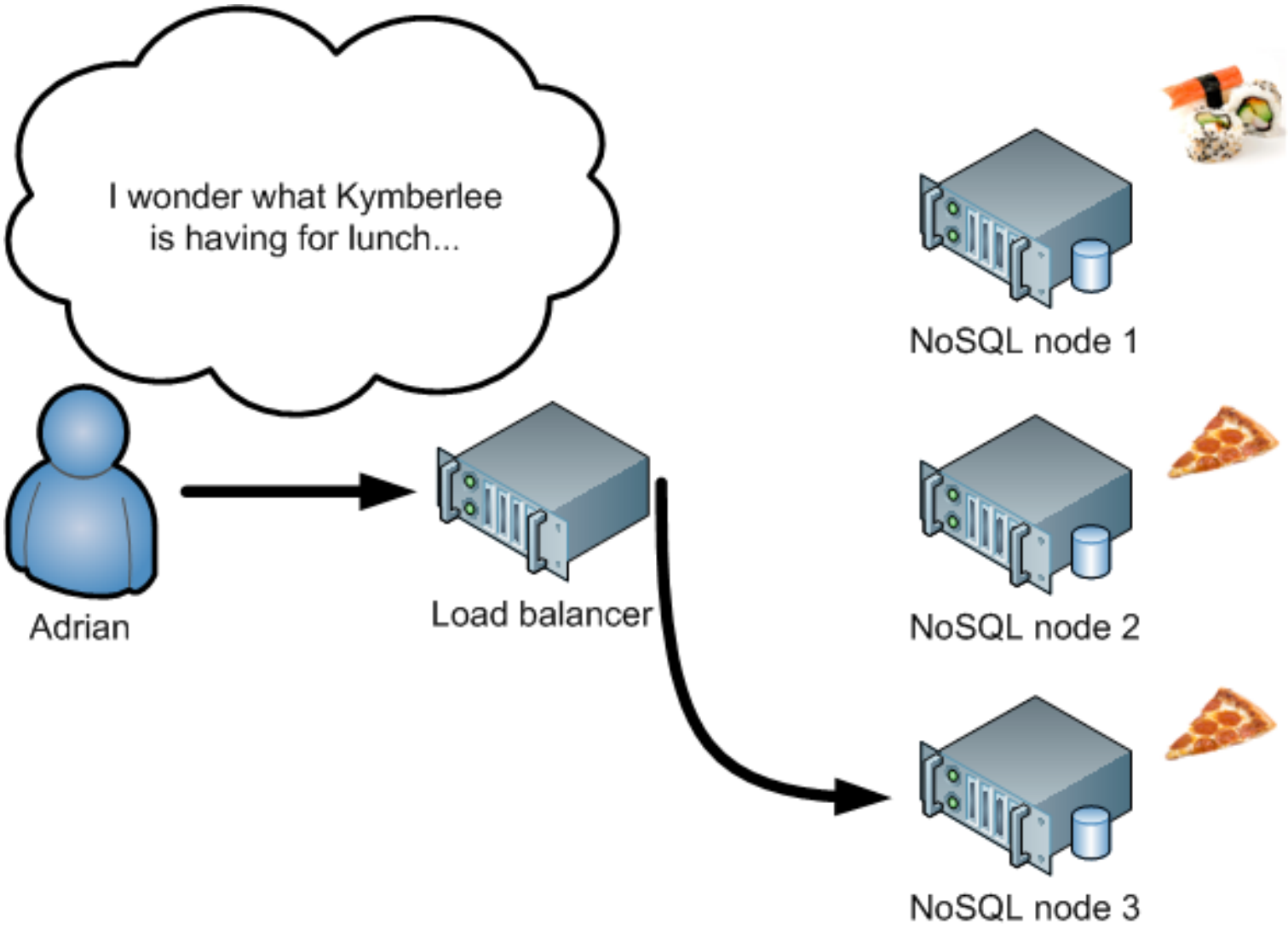
Adobe

Writes don't propagate immediately

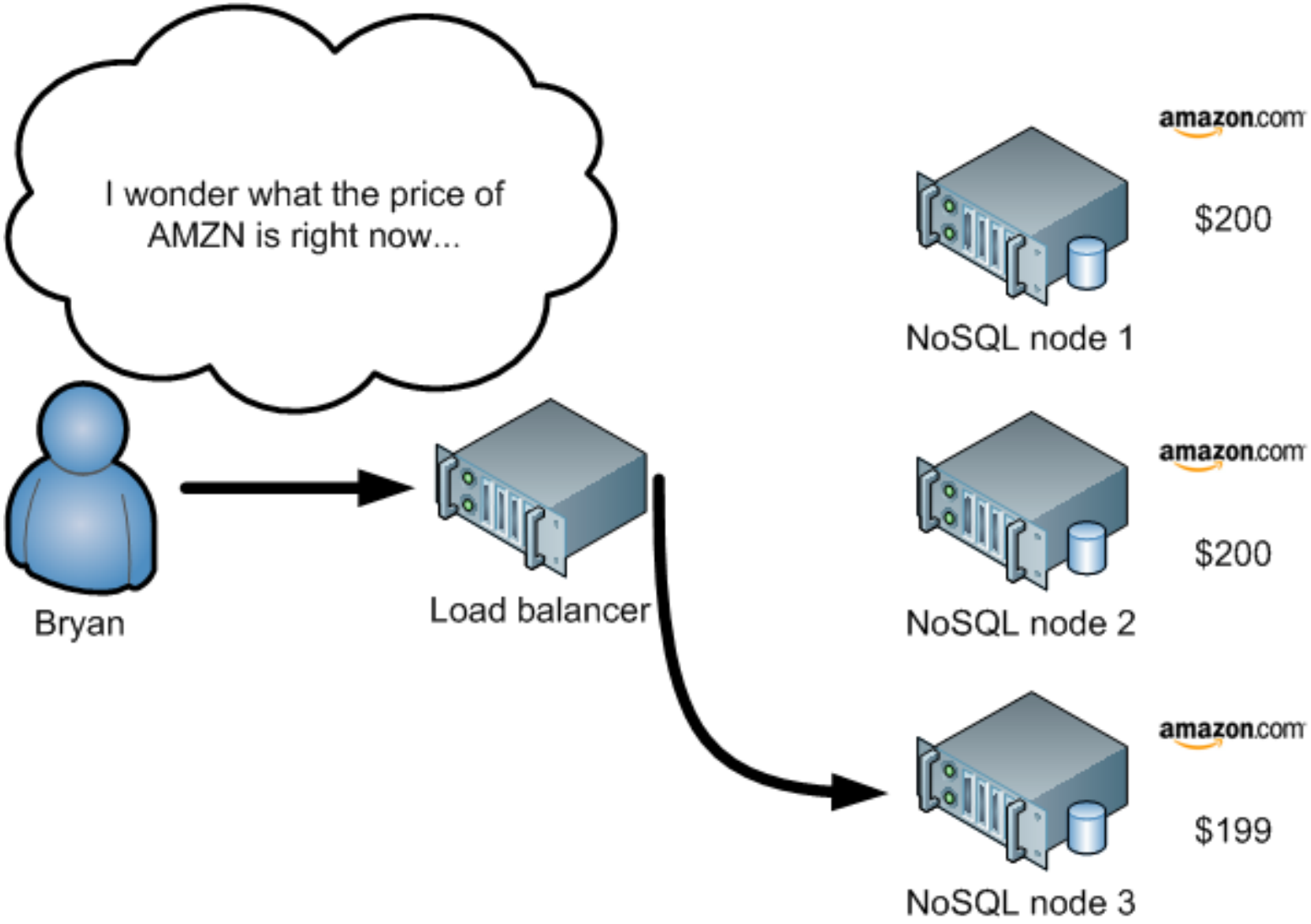


Adobe

Reading stale data



Reading stale data – a more serious case



Agenda

Eventual Consistency

REST APIs and CSRF

NoSQL Injection

SSJS Injection

Adobe

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…

Database	Default Port
MongoDB	27017 28017 27080
CouchDB	5984
Hbase	9000
Cassandra	9160
Neo4j	7474
Riak	8098

Port Scanning Demo

Port scanning

- If an attacker finds an o… already won…

Databa…	… Port
MongoDB	27017 2801… 27…
CouchDB	
Hbase	…0
Cassandra	…9160
Neo4j	7474
Riak	

Adobe

REST document API examples (CouchDB)

Retrieve a document

```
GET /mydb/doc_id HTTP/1.0
```

Update a document

```
PUT /mydb/doc_id HTTP/1.0
{
 "album" : "Brothers",
 "artist" : "The Black Keys"
}
```

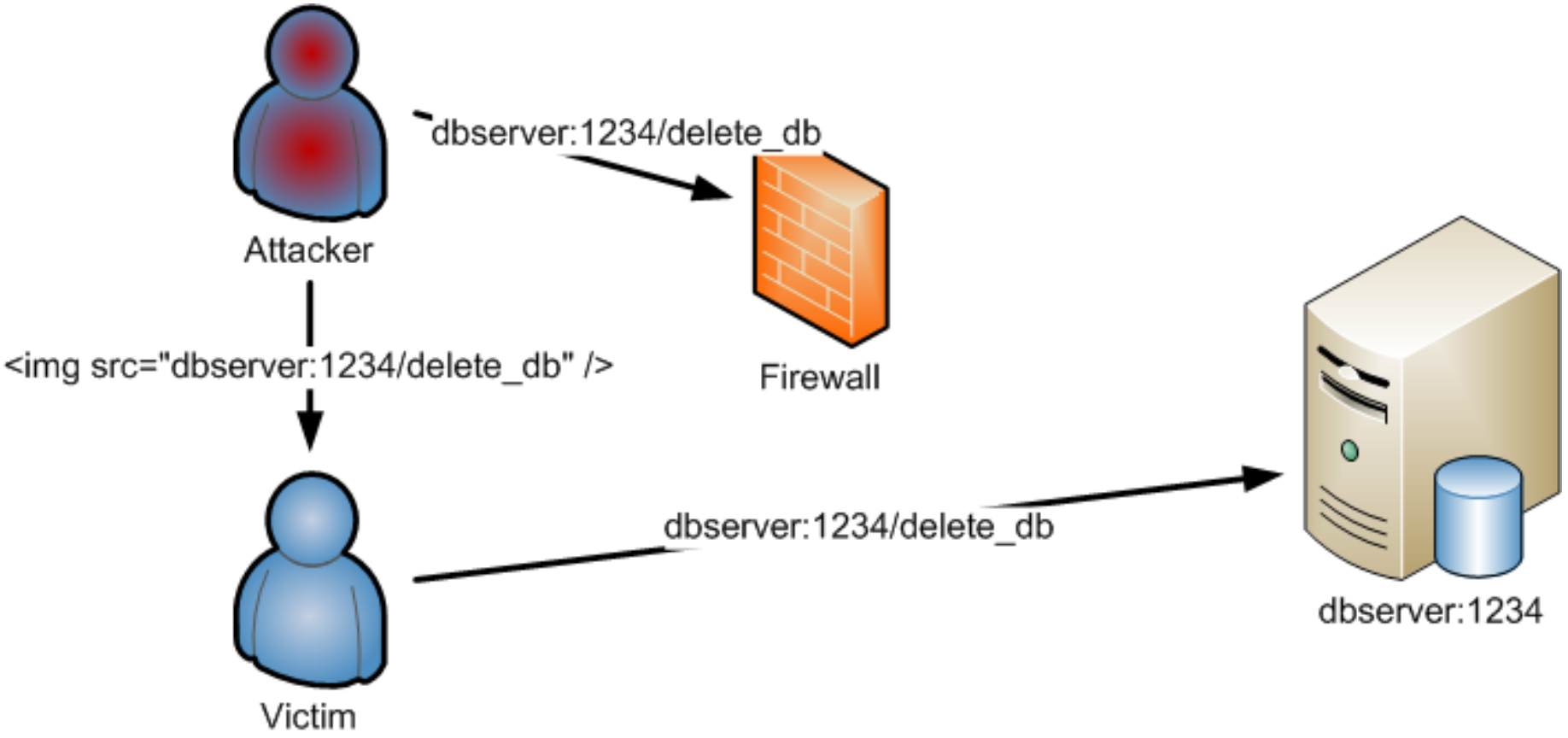
Create a document

```
POST /mydb/ HTTP/1.0
  {
   "album" : "Brothers",
   "artist" : "Black Keys"
  }
```

Delete a document

```
DELETE /mydb/doc_id?
 rev=12345 HTTP/1.0
```

Cross-Site Request Forgery (CSRF) firewall bypass



REST document API examples (CouchDB)

Retrieve a document

```
GET /mydb/doc_id HTTP/1.0
```

Update a document

```
PUT /mydb/doc_id HTTP/1.0
{
 "album" : "Brothers",
 "artist" : "The Black Keys"
}
```

Create a document

```
POST /mydb/ HTTP/1.0
  {
   "album" : "Brothers",
   "artist" : "Black Keys"
  }
```

Delete a document

```
DELETE /mydb/doc_id?
 rev=12345 HTTP/1.0
```

Traditional GET-based CSRF

```
<img src="http://nosql:5984/_all_dbs"/>
```

- 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

```
<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

```
<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

Adobe

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

Adobe

NoSQL 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

```
find( { 'artist' : 'The Black Keys' } )
```

- In PHP, you do this with associative arrays

```
$collection->find(array('artist' => 'The Black Keys'));
```

- This makes injection attacks difficult
- Like parameterized queries for SQL

MongoDB and PHP

- 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

Adobe

$where queries

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

```
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

Adobe

Agenda

Eventual Consistency

REST APIs and CSRF

NoSQL Injection

SSJS Injection

Adobe

Browser war fallout

- 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)

Adobe

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

Adobe

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

Adobe


Adobe