Securing Single Page Applications with Token Based Authentication

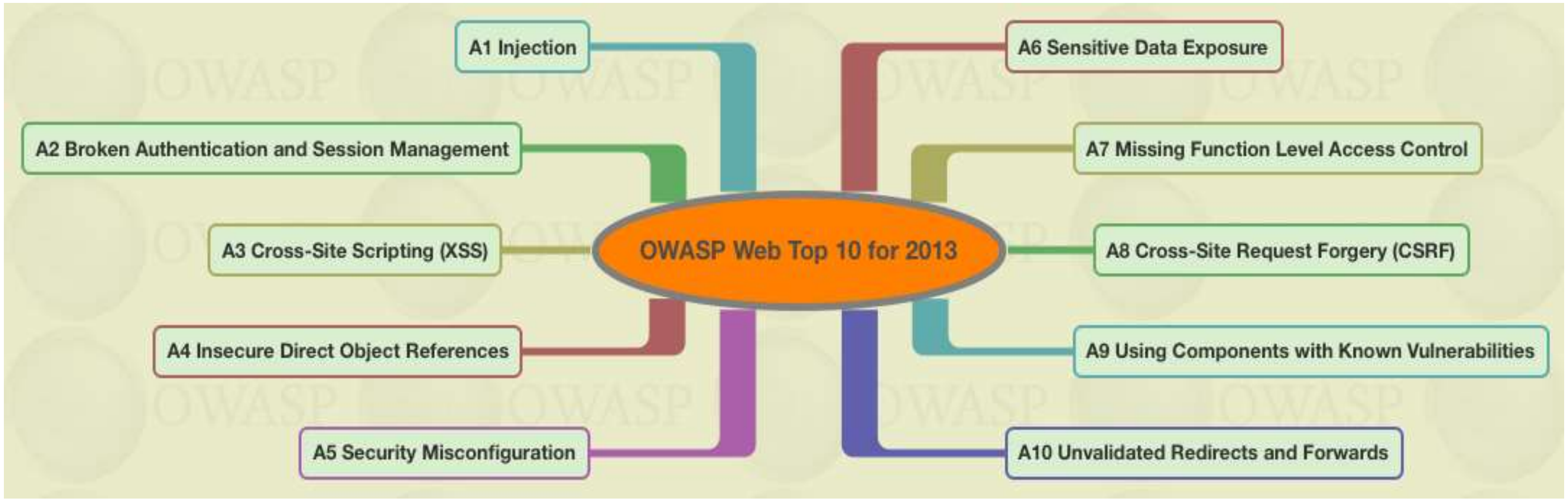
Stefan Achtsnit

WeAreDevelopers Conference
April 2016

Outline

- Single Page Application Security
- Token Based Authentication
    - Client Tokens
    - JSON Web Token format
- Web Application Security 101
    - Cross-site scripting
    - Cross-site request forgery -> Double submit cookie
- Implementation Challenges
    - Who should be in charge of the authentication token – JS client or Browser?
    - How to revoke issued tokens?

Single Page Application Security


OWASP Web Top 10 for 2013
A1 Injection
A2 Broken Authentication and Session Management
A3 Cross-Site Scripting (XSS)
A4 Insecure Direct Object References
A5 Security Misconfiguration
A6 Sensitive Data Exposure
A7 Missing Function Level Access Control
A8 Cross-Site Request Forgery (CSRF)
A9 Using Components with Known Vulnerabilities
A10 Unvalidated Redirects and Forwards

- no sensitive information like keys on clients
- prevent malicious code from running in your application (Cross-site scripting – see upcoming slide)
- secure user credentials (Man-in-the-middle attacks)
- strive for consistent authorization, i.e. UI rendering based on same rules as used for API (Client Tokens)

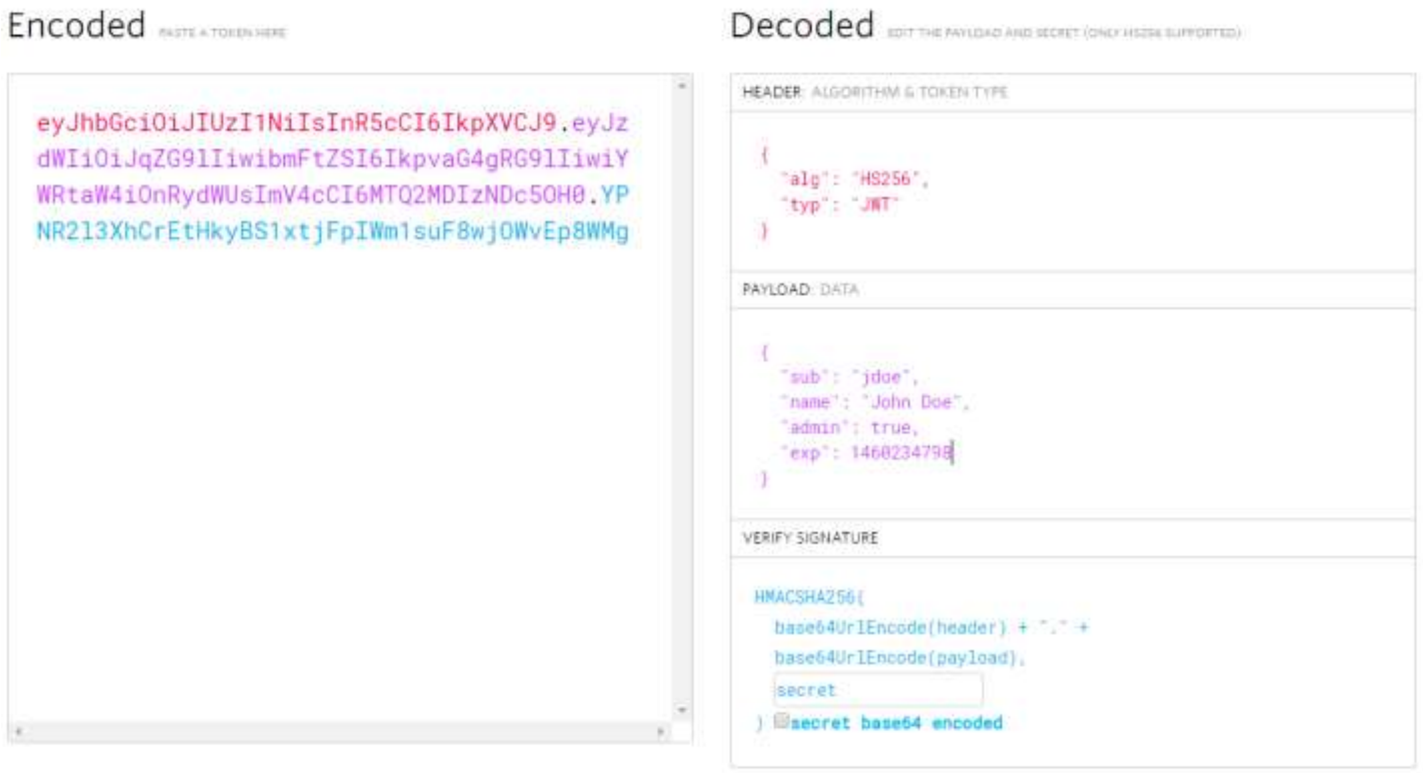
Client Tokens

- self-contained set of claims that assert an identity and a scope of access that can be shared (no silo)

```
{
  "subject": "jdoe",
  "name": "John Doe",
  "admin": true,
  "expiration": 12-04-2016 23:55 UTC
}
```

- signed and optionally encrypted
- flexible, e.g. extend with "issuer“ claim for verification, add application specific ACLs
- stateless (token revocation - see upcoming slide)

JSON Web Token (JWT) format



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz
dWIiOiJqZG9lIiwibmFtZSI6IkpvaG4gRG9lIiwiY
WRtaW4iOnRydWUsImV4cCI6MTQ2MDIzNDc5OH0.YP
NR213XhCrEtHkyBS1xtjFpIWm1suF8wjOWvEp8WMg
```

Decoded EDIT THE PAYLOAD AND SECRET (ONLY HS256 SUPPORTED)

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "jdoe",
  "name": "John Doe",
  "admin": true,
  "exp": 1460234798
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) ☐secret base64 encoded
```

- de facto standard token format
- sign with strong key and always verify token
- encrypt for sensitive information (JSON Web Encryption)
- security considerations as for session identifiers

Cross-site scripting (XSS)

- attacker pushes malicious JS code into application
- canonical example: script tag in user comment
- various categories like stored, reflected or DOM-based attacks
- OWASP - XSS Prevention Cheat Sheet
  - always validate user input and escape everything
  - be careful with dynamically loaded JSON, CSS, HTML templates,…
- Content Security Policy (CSP)
- 3rd party JS libraries are still problematic, everything accessible by JS like tokens stored in web storage may be exposed -> possible alternative: use cookie storage with HttpOnly flag for tokens

Cross-site request forgery (CSRF)

- browser automatically sends cookies set on a given domain with every request made to that domain (regardless of where those requests originated)

```
mybank.org - cookie authentication without CSRF protection

1) HTTP GET request
<img href="https://mybank.org/payments?recipient=JohnDoe&amount=100" />

2) HTTP POST request
<script type="text/javascript">
  $(document).ready(function() {
    window.document.forms[0].submit();
  });
</script>
<div style="display:none;">
  <form action="https://mybank.org/payments" method="POST">
    <input name="recipient" value="JohnDoe" />
    <input name="amount" value="100" />
  <form>
</div>
```

- OWASP - CSRF Prevention Cheat Sheet
  - no side effects with HTTP GET
  - use Double submit cookie with HTTP POST for SPAs

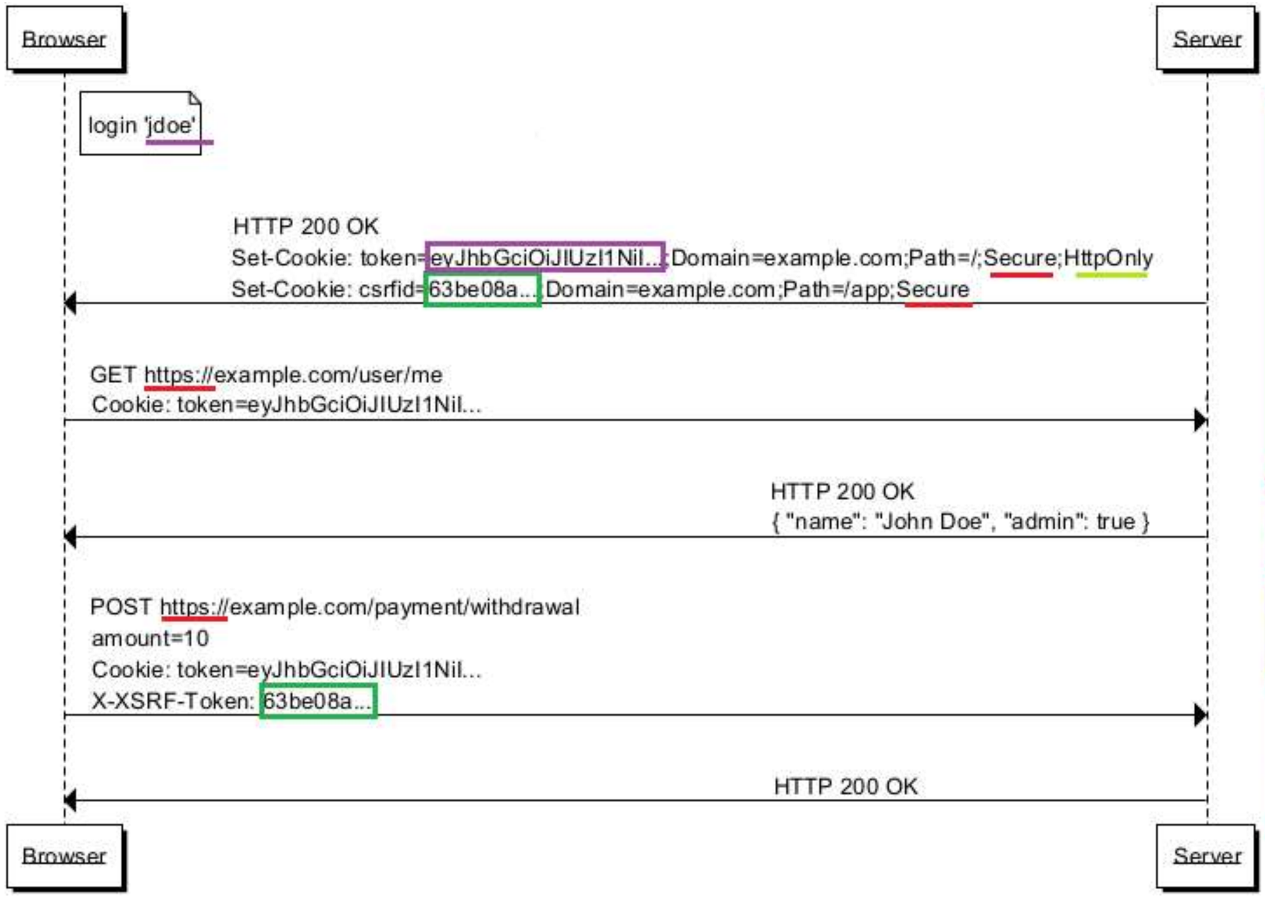
Double submit cookie

- create random value during authentication process
- add this value as additional claim to the provided client token

```
e.g. {
  "subject": “jdoe",
  "name": "John Doe",
  "admin": true,
  "expiration": 12-04-2016 23:55 UTC,
  "xsrfToken": 63be08af-0264-47af-b2b9-6d56e8f6428a
}
```

- put this value in further authentication cookie (with HttpOnly cookie flag set to false), so that JS client can read value and forward it with corresponding HTTP header in subsequent requests
- server side API code can compare client token claim with submitted HTTP header value during token verification (stateless!)
- protection based on same-origin policy for cookies - only JS code running on the origin domain can read this second cookie!

Request Flow Example



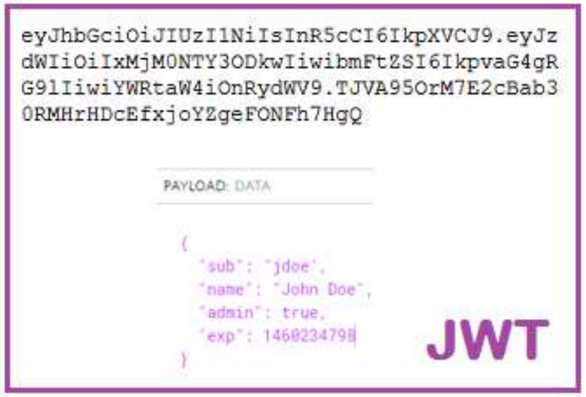
```
HTTP 200 OK
Set-Cookie: token=eyJhbGciOiJIUzI1NiI...;Domain=example.com;Path=/;Secure;HttpOnly
Set-Cookie: csrfid=63be08a...;Domain=example.com;Path=/app;Secure

GET https://example.com/user/me
Cookie: token=eyJhbGciOiJIUzI1NiI...

HTTP 200 OK
{ "name": "John Doe", "admin": true }

POST https://example.com/payment/withdrawal
amount=10
Cookie: token=eyJhbGciOiJIUzI1NiI...
X-XSRF-Token: 63be08a...

HTTP 200 OK
```



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz
dWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gR
G9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab3
0RMHrHDcEfxjoYZgeFONFh7HgQ
```

PAYLOAD: DATA

```
{
  "sub": "jdoe",
  "name": "John Doe",
  "admin": true,
  "exp": 1460234798
}
```

JWT

HTTPS only (MITM)

Hide from JS (XSS)

Double Submit Cookie (CSRF)

Who is in charge of the token?

	JS client	**Browser**
Transmission	`Authorization: Bearer <JWT>` manual coding effort, only when necessary, works with any domain (Cross-Origin Resource Sharing)	`Cookie: token=<JWT>` automatically sent, overhead when not necessary, not possible across domains i.e. with external APIs
Storage	various options, e.g. • web storage (accessible only from storing subdomain, 5MB limit) • cookie storage (accessible from multiple subdomains, 4KB limit)	cookie storage
MITM	SSL must be managed by code	Secure cookie flag forces SSL
XSS	manual coding effort	implicit with HttpOnly cookie flag to prevent JS access
CSRF	not applicable	manual coding effort (e.g. Double submit cookie)

How to revoke tokens?

	Renewal approach	**Blacklisting approach**
Characteristic	stateless	stateful
Mechanism	two kinds of tokens used: Access Tokens as usual with a short expiration and Refresh Token with a longer expiration for the renewal of Access Tokens when they expire (OAuth2!)	Identifier ("jti" claim) added to Access Token and checked against blacklist during token verification
Consequence	client flow implementation not trivial, additional server logic only within authentication API	transparent for client, blacklist must be available to all server APIs

Recommendation

- follow a token based approach with JWT
- avoid cross-domain architecture if possible
- CSRF protection is easy to get right, XSS protection is easy to get wrong
  - store token in cookie with HttpOnly (XSS) and Secure flag (MITM)
  - use Double submit cookie (CSRF)
- start with a simple token revocation mechanism
- do your homework and don't reinvent the world
  - Open Web Application Security Project (OWASP)
    - https://www.owasp.org/index.php/Top_10_2013
    - https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet
    - https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet
  - JSON Web Token
    - https://jwt.io

Thank you!