How to secure your web-app

At least a little bit

Denis Rechkunov

Disclaimer

I’m not a security expert!

So please check every word I say.

What will we cover

- Basic definitions
- Attack surface in web-apps
- Attack vectors
    - Common vectors
    - Demos
    - Measures to reduce the attack surface
    - Scary stories from my experience

Some definitions first

- UGC – User-Generated Content
- Hacker – a person who can figure out how things work very quickly
- Black hats – hackers who use their findings for bad
- White hats – hackers who are security researchers and report problems
- Attack vector – a method how an app can be exploited
- Attack surface – basically any part of your software exposed to the world

Attack surface in Web-apps

Web-apps have one of the largest attack surfaces because:

- Unless it’s a static web-site, they’re meant to interact with the user
- Nowadays they’re mostly running in a browser (JavaScript)
- Browsers and web-standards (which are many) are not under your control when your build an app but, in fact, they’re a huge part of your app

Attack Vectors

Sensitive Data Exposure

- Confidential data can be served as a part of the API response (N26 was hacked by researcher Vincent Haupert in 2016)
- GitHub and Twitter were logging plain text passwords until May 2018. **Any engineer had access to your password until then** *or maybe still has*
- Attacker is able to:
  - Impersonate a user talking to the tech support
  - Reset the password
  - Steal identity
- To prevent:
  - **NEVER** log any sensitive data
  - **NEVER** propagate error messages to the user directly
  - **NEVER** include any information to the response if it’s not supposed to be displayed

Sensitive Data Exposure

How to debug errors with external users then?

- Use general error messages
- Encrypt all details into a token which users can attach when they file an issue. Google uses this approach (or at least used when I got 500 last time)

SQL injection

- One of the oldest attack vectors
- Can be performed via input fields or URL
- Attacker is able to:
  - Bypass authentication
  - Destroy your database
  - Get access to any data in the database
- To prevent:
  - **NEVER** build an SQL query as a string manually
  - **ALWAYS** use libraries and frameworks that know how to escape query parameters

More https://www.owasp.org/index.php/SQL_Injection

Demo and the story

SQL injection

Examples:

```
“SELECT * FROM users
WHERE username = ‘“+username+“’ AND password = ‘“+password+“‘;”
```

Using `‘ OR 1=1` as a password will form:

```
SELECT * FROM users
WHERE username = ‘some’ AND password = ‘’ OR 1=1;
```

which is always true

SQL injection

Examples:

```
“SELECT * FROM users
WHERE username = ‘“+username+“’ AND password = ‘“+password+“‘;”
```

Using `‘; DROP TABLE users` as a password will form:

```
SELECT * FROM users
WHERE username = ‘some’ AND password = ‘’; DROP TABLE users
```

It will destroy your `users` table, hopefully you have a backup

SQL injection

Examples:

```
“SELECT * FROM users WHERE user_id = ‘“+uuid+“’;”
```

Using `‘ OR 1=1` as UUID will form:

```
SELECT * FROM users WHERE user_ID = ‘’ OR 1=1;
```

It can expose information about all the users (depends how you render the result)

Cross-site scripting (XSS)

- Usually performed via UGC or URL (in old browsers) that contains JavaScript
- Attacker is able to:
  - Steal any information associated to the web-site and stored in your browser: cookies, local storage content, globally stored data in JavaScript, etc
  - Perform any action on your behalf
    - HTTP requests
    - Form submission
    - Redirect
- To prevent:
  - **ALWAYS** filter/escape all the UGC and URL parameters either on input or output
  - **NEVER** render any UGC as data in script tags or DOM node attributes
  - Protect cookies with `httpOnly` if possible, so JavaScript has no access to it

Demo and the story

Cross-site scripting (XSS)

Examples:

- Data rendered directly from URL with no escaping/filtering
- UGC is not escaped with HTML entities (`<` as `&lt;`, `>` as `&gt;`, etc)
- UGC is escaped only for known tags like `<script>`, `<div>`, `<span>`, etc.
- UGC is rendered inside a script tag and contains `</script>`

Anchor abuse

- Performed via rendering links where `href` is UGC
- Attacker is able:
  - To perform XSS attack using e.g. `javascript:alert(‘you\’re hacked’)`
  - To steal your current URL including query via `document.referrer`
  - Get access to the previous page via `window.opener` and even make changes (made public by Mathias Bynens in 2016)
- To prevent:
  - Allow only `http:` and `https:` as URL schema
  - Redirect to your proxy first and then to the external URL striping all the data
  - **NEVER** store sensitive data (like tokens) in URL
  - **ALWAYS** set `rel="noopener noreferrer"` for external links

Demo and the story

Anchor abuse

Examples:

- Feed with UGC where users post links with `javascript:` as schema
- URL contains a security token in query and the user clicks a UGC link
- A malicious web-site changes address of a previous tab, user tricked into submitting login and password on a fake page

Cross Site Request Forgery (CSRF)

- Can be performed via HTML forms or even embedded images
- Attacker is able to execute an action on user’s behalf (e.g. bank transfer)
- To prevent:
  - **NEVER** change the state on GET requests, these request can be triggered by an image
  - **ALWAYS** use a right HTTP method
  - **ALWAYS** configure Cross-Origin Resource Sharing (CORS) properly
  - **ALWAYS** use a random CSRF token which is generated by the website every time the user loads the page and expires soon

Demo and the story

Cross Site Request Forgery (CSRF)

Examples:

- An API endpoint that deletes a user using GET instead of DELETE method. Attacker embeds a malicious tag

  ```
  <img src=”https://admin.website/users?action=delete&id=42”>
  ```

- Attacker puts a malicious form on a website that submits to another website where the user logged in

  ```
  <form action=”https://admin.website/users”>
    <input type=”hidden” name=”username” value=”batman”>
    <input type=”hidden” name=”password” value=”iambatman”>
    <button type="submit">Donate to kittens!</button>
  </form>
  ```

Cross Site Request Forgery (CSRF)

Simple CSRF token (Encrypted Token Pattern):

- Encrypt `{ “timestamp”: #########, “userId”: ### }` with AES256
- Put the generated token in each form rendered on server

```
<form action=”https://admin.website/users”>
  <input type=”hidden” name=”csrf” value=”randomtokenvalue”>
                         ...
  <button type="submit">Submit</button>
</form>
```

- Decrypt, deserialize and validate user ID and lifetime of the token when the form is submitted

iframe-related vectors

- Performed via embedding your web-app in an `iframe` on the attacker’s website. Old browsers could leak key press events from the `iframe`.
- Attacker is able:
  - To record everything user type inside the `iframe` (your web-app) in IE
  - To trick your users that their website is a part of yours (e.g put additional fields)
- To prevent:
  - Don’t allow to embed your web-app in an iframe/frame
    - Header `X-Frame-Options: deny`
    - Content Security Policy (CSP): `frame-ancestors 'none'`

Unrestricted File Upload

- Can be performed via any file upload endpoint using a malicious payload
- Attacker is able:
    - To crash the server that handles the request with “Out of Memory”
    - Potentially execute malicious code
- To prevent:
    - **ALWAYS** limit the request body size either in your app or in a proxy like NGINX
    - **NEVER** evaluate uploaded code on server

More https://www.owasp.org/index.php/Unrestricted_File_Upload

Unrestricted File Upload

Examples:

- “Bare” node.js server that handles uploads directly with `.on(‘data’,chunk=>{...})` and `.on(‘end’, ()=>{...})` events
- Evaluating uploaded JavaScript or Bash on server

Other attack vectors

- Brute forcing – rate limiting, captcha
- Phishing – nothing we can do, users need to look at URLs. User education.
- Broken Authentication – don’t reinvent the wheel, use existing auth standards
- Security Misconfiguration – sometimes the default config is insecure
- Tokens sent via email don’t expire – email is insecure, can’t be trusted, so everything sent via email must expire

Preventive Measures

Cross-Origin Resource Sharing (CORS)

- Set of request/response headers supported by browsers since ~2009
- Features:
  - Limit where your HTTP API can be accessed from
  - Limit what headers are allowed
  - Limit what HTTP methods are allowed
- Your HTTP API must support `OPTIONS` requests with following headers:
  - Access-Control-Allow-Origin
  - Access-Control-Allow-Methods
  - Access-Control-Allow-Headers
  - etc.

More https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

Content Security Policy (CSP)

- Header `Content-Security-Policy` supported by browsers since 2014-2015
- Features:
  - Limit where to download media: fonts, styles, scripts. You can even sign or limit to SHA
  - Limit where to make AJAX requests
  - Limit what frames are allowed and if it’s allowed to embed the web-app
  - Limit where to submit forms on the page
  - Sandbox mode – disable/enable JavaScript, modals, popups, forms, etc
- Crucial for high-security parts of your app like a login form

More https://content-security-policy.com

What to read/watch

- Open Web-Application Security Project https://www.owasp.org – basically wikipedia-like source about web-security
- Top 10 vulnerabilities 2017
- Content Security Policy https://content-security-policy.com
- Cross-Origin Resource Sharing https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
- Vincent Haupert – Shut Up and Take My Money! – N26 vulnerabilities https://www.youtube.com/watch?v=KopWe2ZpVQI
- Demos https://github.com/rdner/security-nightmare

“Trust is not a renewable resource”

Matthew Green

Thank you!
Be safe.