Security Hackathon

OWASP Top 10

- OWASP stand for Open Web Application Security Project
- They publish a Top 10 list every couple year of the most common vulnerabilities
- It's not just a list, but also a detailed description of each of the problem, and how to mitigate and avoid them
- Definitely worth a read: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

OWASP Top 10 2017

- The latest version is from 2017 listing the following common cases:
  - Injection
  - Broken authentication
  - Sensitive Data Exposure
  - **XML External Entities**
  - Broken Access Control
  - Security Misconfiguration
  - Cross Site Scripting (XSS)
  - **Insecure Deserialization**
  - Using components with known vulnerabilities
  - **Insufficient Logging and monitoring**
- The bold ones above are new entries replacing the following dropped ones since 2013:
  - CSRF
  - Unvalidated redirects

A1:2017-Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

A4:2017-XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
A5:2017-Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
A6:2017-Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

A7:2017-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A8:2017-Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

A9:2017-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10:2017-Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

Admin #1
Weak Password

Username: admin
Password: hunter

Hunter is the 27th most common password. I chose the 27th, as most lists you can google only go until 25, and the 26th one was a swear word

How to hack: simple brute force, there's plenty of tools out that try

Mitigation: Disallow simple passwords, check passwords when set so they're not in the vulnerable or leaked ones. Databases like https://haveibeenpwned.com/ list common passwords and passwords that have been leaked. Also rate limiting and captchas can help mitigate brute force attacks

Admin #2
Leaked password

How to hack: Go to https://github.com/sztupy/security-hackathon and check the .env file

Mitigation: Always check what you are pushing to GitHub. Tools like https://www.gitguardian.com/ can help automatically check and notify users if they accidentally upload something that looks like a secret. Once secrets are uploaded the only safe thing to do is revoke access to that key completely

Admin #3
Hidden field on the client side

OWASP #2: Broken Authentication

There was a hidden input called admin on the form

How to hack: use GET /guest?admin=1

Mitigation: Never trust client input

Admin #4
Hidden url

OWASP #2: Broken Authentication

How to hack: GET /admin once logging in as guest

Mitigation: Always have proper authorization checks before allowing access to a controller

Admin #5
Exiting sandbox

How to hack: GET /avatar?url=../../../etc/passwd

Mitigation: Never trust user input. Always make sure you have proper sandboxing set up when requesting local resources.

Admin #6
SQL injection

OWASP #1 Injection

How to hack: use `' OR ''='` as the username

Mitigation: Never trust user input, always make sure to follow best practices when using libraries accessing external resources where user inputted data is used

Admin #7
Unsecured ssh access

OWASP #6: Security misconfiguration

How to hack: call ssh admin@<url>; Password is admin

Mitigation: Disallow password auth. Make sure to use strong keys. Make sure to have IP whitelisting for access

Admin #8
Unsecured postgres instance

OWASP #6: Security misconfiguration

How to hack: psql -h <url> -U postgres

Mitigation: Make sure all backend services are properly secured.

Admin #9
JWT validation bug

OWASP #9 Using components with known vulnerabilities

How to hack: Obtain JWT, change algorithm to HS256, change the contents to give you access, and use the public key as encryption key

Mitigation: Make sure to follow best practices, keep up to date with libraries

Read more: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

Admin #9
JWT validation bug

JWT is composed of three parts: header, payload and signature.

Header contains some metadata around the JWT including what kind of authentication is used for the JWT itself. The most common authentication mechanisms are Hash based HS256 and Private/public key based RS256

HS256 is fairly simple, you use the same secret key for creating JWT and for validating if a JWT is not tampered with

RS256 is a bit more complex, you use a private key to generate JWT, but then use a public key for validating it was not tampered with

Admin #9
JWT validation bug

The main benefit of RS256 however is that you can allow anyone to check that your JWT is valid, as validation is done using a public key. This means you can have a very secure box which generates JWTs, and validation of that JWT can be done locally using publicly available information, and you won't need to know the secret key.

So what's the problem?

Admin #9
JWT validation bug

A lot of libraries had an API similar to the following:

CreateJWT(algorithm, payload, secret)

VerifyJWT(jwt, secret)

Note the lack of algorithm on the verify part, as that is contained in the jwt header anyway. The secret variable is either the AES key for HS256 on both create and verify, or the private key on create and the public key on verify for RS256

Admin #9
JWT validation bug

A lot of libraries had an API similar to the following:

CreateJWT(algorithm, payload, secret)

VerifyJWT(jwt, **secret**)

Note the lack of algorithm on the verify part, as that is contained in the **jwt header** anyway. The secret variable is either the AES key for **HS256** on both create and verify, or the private key on create and the **public key** on verify for RS256

Admin #9
JWT validation

So what if someone changed the algorithm in the header to HS256 and used the public key (which is public) to sign it?

VerifyJWT(jwt, public key) would return TRUE

So what did JWT library authors do?

VerifyJWT(algorithm, jwt, secret)

Made the algorithm part of the API

Admin #10
Padding oracle attack

OWASP #9 Using components with known vulnerabilities

How to hack: Use a padding oracle attack on the token, change the value and re-encrypt it. Tools like https://github.com/iagox86/Poracle can help

See slide #2

Bonus points
Social Engineering attack

Anyone trying to send me a phishing email or simply to ask for the password directly gained a bonus point