Unit 11 –

Web security is rapidly and constantly evolving.

We need to therefore be especially aware of some of the big-picture principles.

Principles of authenticating users – hashing passwords, utilizing bcrypt library to store passwords…

1. Authentication
   1. Making sure someone is who they say they are –
   2. Validating their credentials
   3. Managing a session on the user’s browser.
      1. Different services have different session lengths
      2. Think FB vs BofA
2. Authorization is separate matter from authentication.
   1. Authorization is the **process** by which we control **which resources** **that an authenticated user can access.**
3. HTTP vs HTTPS
   1. HTTPS – “Secure” HTTP protocal.
   2. All data is encrypted, rendering it useless to anyone except the intended recipient (server).
   3. Secure connections are established through
      1. SSL or TLS protocols.
         1. Secure socket layer
         2. Transport layer security
      2. When we deploy our apps, **we want https://**
      3. Google
4. Authentication is to **protect users.**
   1. We encourage them to follow good practices but we can’t guarantee it.
   2. Users will reuse passwords, reuse emails, etc.
   3. So user passwords are typically not random or unique, so these are security vulnerabilities.
   4. To mitigate this, the basic practice is **password hashing.**
   5. **NEVER STORE PASSWORDS AS CLEARTEXT.**
   6. Cleartext/plaintext is humanly readable.
   7. **Ciphertext** is obscured by hashing/encoding.
      1. **YOU SHOULD NOT BE SENDING THE USER THEIR PASSWORD BACK IN AN EMAIL, THAT SHIT’S RIDICULOUS IN 2023.**
   8. Even if data is leaked, encrypted/hashed data is safer.
5. Encoding
   1. The process of transforming input data from one format to another.
   2. This can stop malicious code from being executed elsewhere.
   3. Weak security
6. Encryption
   1. Transforming the input data so that only the intended recipient can interpret the original message.
7. Hashing
   1. Hashing transforms an input into a new string that cannot be reverse engineered.
   2. Hashing is a one-way transformation.
   3. No matter how big that string is, it will always be consistently morphed into the same thing.
   4. ‘Hello’ or the entirety of ‘war and peace’ should still be hashed into the same string every single time, each 22 characters (or whatever).
8. Properties of hash functions
   1. **Hash functions return the same hash with the same input**
   2. **Hash functions are one-way.**
   3. We use bcrypt for this.
   4. What is bcrypt?
9. Bcrypt
   1. This is a hashing algorithm that uses a randomly generated string of chars (a “salt”) to make every stored password unique.
   2. When a user sets a password, “salt” is added to the end.
   3. And the resulting string repeatedly hashes the password+salt, determined by a **work factor** parameter that indicates how many times the hash function should run.
   4. This way, two users with the same password will have different stored passwords.
   5. When a user sets the password (whereby the salt is generated, concatentated,and the results hashed an X number of times),
      1. A string with the X number time, the salt, and the resulting hash is stored.
   6. When a user logs in a second time, the salt is read from the stored string and added to the password
      1. Then it takes the number of times to hash the string
      2. And is able to generate the identical hash and check the stored salt+hash in their database.
   7. Why do we hash something 2^10 (or whatever large number) amount of times?
      1. A hacker who knows the hashing algorithm (defined by bcrypt’s 2a or whatever tag at the beginning) and the salt (by maybe guessing the right number of characters), will still need to figure out how many times the hash needs to be run. So making this number larger makes brute-force efforts.

bCrypt demo pseudocode:

*// require in bcrypt library*

*// define userpass*

*//define workfactor*

*//create useracct*

*//run hash method on bcrypt, passing in password and workFactor*

*// returns a promise. "then" take the hash and we use*

*// fs.writeFileSync to store this in the database.*

*// and catch any errors.*

*// validating passwords*

*// get the inputted passwords*

*// get the storedHash by using fs.readFileSync on the database.*

*// and use .compare method on bcrypt, passing in inputtedPass and storedHash*

*// and use .then to check the result. "result" is a true/false depending on match.*

1. Security Aside:
   1. Dictionary Attacks
      1. Try common passwords. Works better for targeting single users.
      2. Not so common anymore, with common security practices (like limiting login attempts within a timeframe).
   2. Brute-force attacks.
      1. They can pre-hash commonly used passwords.
      2. Random salts are used to protect against this.
2. User Enumeration
   1. Get user information based on clues accidentally left behind by our authentication process.
   2. For example, we want to say username/password does not exist rather than “user does not exist” – this is because
   3. Case Study:
      1. Ashley Madison was attacked
      2. Login attempts made with a bunch of emails and intentionally wrong passwords.
      3. AM did not reveal if the email didn’t exist.
      4. But the HASH algo was only attempted for registered emails.
      5. So hackers were able to find registered users by **timing** the difference in the response. The response was faster when the user wasn’t registered.
      6. This was used to generate a list of registered users.
3. Authorization
   1. Always define what access each user has.
   2. Good authorization protocols **clearly define which users are allowed to access which resources.**
   3. Ensure these rules are consistently enforced throughout your application.
      1. For example, just removing something from the HTML page makes things still accessible by using Postman.
   4. Do **not** give users a way to access resources they should not have access to.
   5. **Princple of least privilege** – everyone has as little access as possible all the time.
   6. **Validate permissions on** **every request –** every fetch should check authorization.