Security

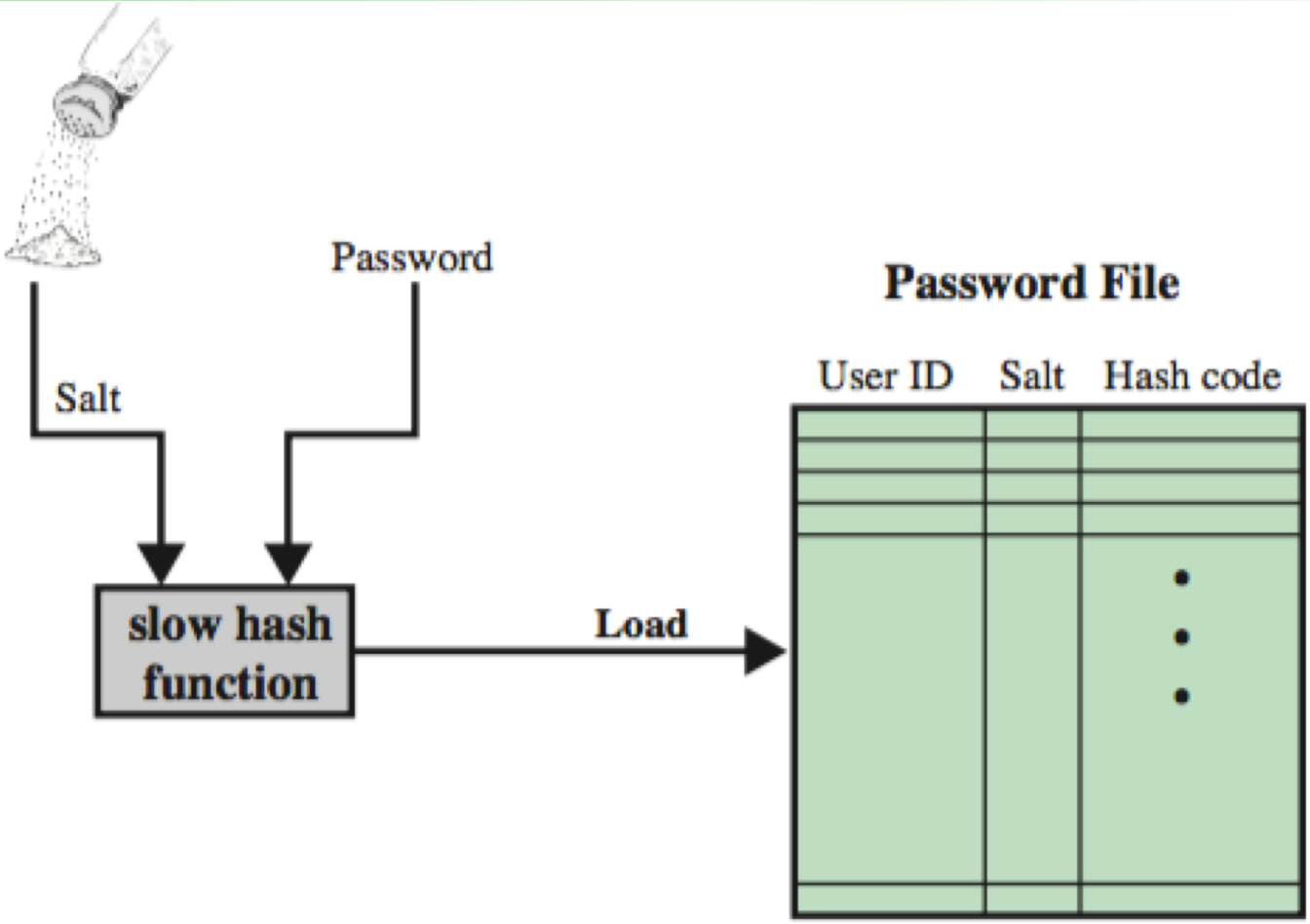
Secure Web Development

– Authentication

Web authentication – credentials

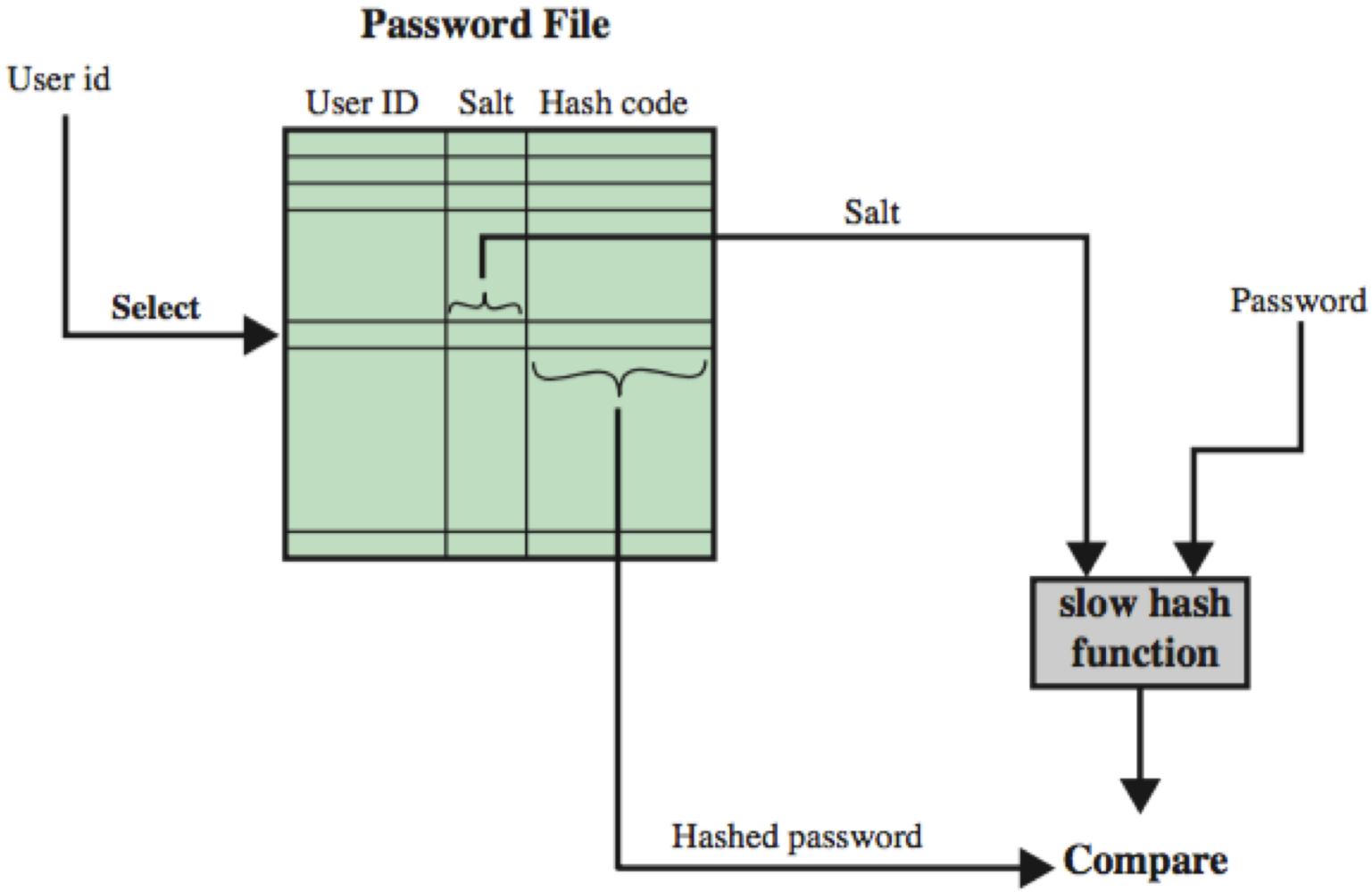
- Do not include credentials in hidden fields, headers, cookies (or code!)
- Passwords/credentials should be stored securely in a centralised location
  - Should only be readable by suitably privileged users
- Passwords should be “salted” and hashed
  - Salting involves appending random bits to each password
  - Salted password is then hashed (i.e. one-way encrypted) for storage
  - Objective is to store something derived from the password that allows an entered candidate password to be checked …
  - … but such that the password cannot be retrieved (by *anybody*, even an administrator)

Loading a new password



- New line in password file contains user id, salt, and output of (slow) hash function

Verifying a password



Purpose of “salt”

- The salt serves several purposes:
  - Frustrates dictionary attacks.
  - Makes reverse lookup of hash value more difficult
    - e.g. using a search engine or *rainbow tables*
  - Prevents duplicate passwords appearing as duplicates in password file
  - Protects users where same password is reused on different systems/sites.

Password hashing & salting in practice - bcrypt

- The ***bcrypt*** module makes it fairly straightforward
  - To create password hash:

```
saltRounds = 10    # this can be tuned to control time to hash
bcrypt.hash(plaintextPassword, saltRounds, function(err, hash) {
  // Store hash in your password DB.
});
```

  - To check entered password:

```
bcrypt.compare(candidatePassword, hash, function(err, res) {
    // res is true if password is correct
});
```

Password hashing & salting in practice - bcrypt

- Even more straightforward with *async/await*
  - To create password hash:

```
hash = await bcrypt.hash(plaintextPassword, saltRounds);
// Store hash in DB instead of password
```

  - To check entered password:

```
const isMatch = await bcrypt.compare(candidatePassword, this.password);
// Check isMatch true or false
```

- Bcrypt is also available for several other languages and frameworks – e.g. jBCrypt for Java

Password strength

- Need to enforce strong passwords
  - Should be resistant to brute force attacks
    - Long, preferably a pass phrase
    - Hard to guess, ideally random
  - User awareness and support
    - Education on non-disclosure
    - Provision of tools (e.g. password manager)
      - People generally cannot remember passwords that are enough to reliably defend against dictionary attacks
  - Audit / check compliance with policy

Password/credentials change policy

- Need secure password change policy
  - Do not use secret questions and answers.
    - Instead, e-mail the user with a time limited activation code and limit account capabilities for 24+ hours
    - Or use out-of-band messaging (e.g. SMS)
  - May require users to change passwords frequently
    - Often mandated by standards/regulations (e.g. PCI DSS)
    - Value of this is debatable
  - "Delete means delete"
    - Not a good idea to keep users' password history
- Credentials such as API keys may have a lifetime and/or associated rotation policy

Multi-factor authentication

- Authentication can be made stronger by requiring a combination of:
  - What you **know** (e.g. password or pass phrase)
  - What you **have** (e.g. SIM card, smartphone app, physical token, certificate file, ...)
  - What you **are** (biometrics)

Session cookies

- Recommended practice is to store authentication tokens in **session** object on server side
  - A session is the time a user spends on a particular visit to a website.
  - Session data maintained by web server in session object to allow for preservation of state across sequence of browser requests
- Store session ID in session cookie
- Make sure framework uses secure session IDs
  - Session IDs should be **long and random** – i.e. impossible to guess

HTTP Authentication

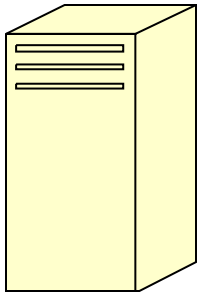
HTTP Authentication

- HTTP provides built-in authentication
- On browsers you get a login prompt

- There are two types of authentication: Basic and Digest

HTTP Basic Access Authentication

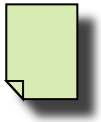
GET /protected/index.html HTTP/1.1



HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Private"

Display
Login panel

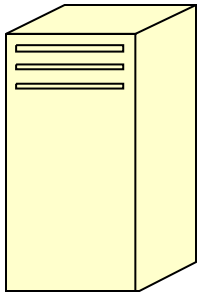
GET /protected/index.html HTTP/1.1
Authorization: Basic JAadf0987awe



Problems with Basic Authentication

- Passwords are easy to intercept
- Repeated Exposure: Password sent with every request
- Passwords are trivial to decode (not encrypted, just Base64 encoded)
- Insecure storage (password cached by browser)
- No logout function

Digest Access Authentication

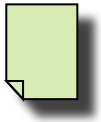


GET /protected/index.html HTTP/1.1

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="Private"
nonce="897sgkjhsadAdsiu"

Display
Login panel

GET /protected/index.html HTTP/1.1
Authorization: Digest username="Alice"
realm="Private" nonce="897sgkjhsadAdsiu"
response="5ijasd9734kuyasds0g"



Challenge and Response

- Challenge (nonce): any changing string
  - e.g. MD5(IP address:timestamp:server secret)
- Response: challenge hashed with the user's name & password and URL of requested page
  - MD5(MD5(name:realm:password):nonce:MD5 (request))
- Server-specific implementation options
  - One time nonce
  - Time-stamped nonce

Digest Advantages over Basic Auth

- Can’t replay the client/server handshake because nonce changes each time
- An intercepted response is valid only for a single web page because the response has the request hashed.

- … still inherently insecure though!