PASSWORD MANAGEMENT

Stephen Schaub

Topics

- Basic Password Issues
- Password Encryption
- Password Reset
- Persistent Authentication (Remember Me)

Basic Password Issues

- How complex should passwords be?
- How should passwords be stored on server?
- When should a user be required to re-authenticate?

Basic Password Recommendations

- From OWASP Authentication Cheat Sheet:
  - Minimum password length should be enforced
  - Permit long passwords
  - Require re-authentication for sensitive features
  - Display non-specific authentication failure messages
  - Design login forms friendly to password managers
- Further reading:
  - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Authentication_Cheat_Sheet.md

Third-Party Authentication Systems

- Handle password management for you
- Examples:
  - OAuth
  - OpenId
  - SAML

Password Encryption

Password Encryption

- Many developers understand that passwords should be stored in the database in encrypted form
  - Why?
- How should a password be encrypted?
  - Standard encryption algorithm?
  - Custom encryption algorithm?

Store Hashed Passwords

- You should assume that your password database may be compromised at some point
- Want to make it hard for the attacker to leverage the encrypted password data
- Key Idea: Rather than storing encrypted passwords, store hashed passwords

Hashing 101

- Hash function
  - Maps input string to fixed-size output
  - Called "one-way" or "trap door" functions because the original value cannot be easily recovered from the hash value

Example:

```
function hash(str) {
  let result = 0
  for (let i = 0; i < str.length; ++i) {
    result = (result + str.charCodeAt(i)) % 256
  }
  return result
}
```

Hash Collision

- Since arbitrarily large inputs hash to fixed-length values, it is possible that two distinct inputs can hash to the same output value
  - This is called a **Hash Collision**
- Some hash functions (ex. MD5) have weak security guarantees and small output sizes
  - Relatively easy to calculate two values with the same MD5 hash
  - How could attackers leverage this?

Cryptographic Hash Functions

- A cryptographic hash function is designed so that a small change to the input string results in a significant change in the output

```
hash("sha256", "");
// e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
hash("sha256", "The quick brown fox jumps over the lazy dog");
// d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
hash("sha256", "The quick brown fox jumps over the lazy cog");
// e4c4d8f3bf76b692de791a173e05321150f7a345b46484fe427f6acc7ecc81be
```

Node.js Cryptographic Hash

- The built-in crypto module provides cryptographic hash functions

```
var crypto = require('crypto')

function hash(algorithm, value) {
  return crypto.createHash(algorithm).update(value).digest('hex')
}
```

Hashing Passwords: Example #1

- User registers password

```
var crypto = require('crypto')
pwd = req.body.password
var hashedPwd = hash('sha256', pwd)
// store hashedPwd in user table in database
```

- Login verification:

```
pwd = req.body.password
var hashedPwd = hash('sha256', pwd)
// compare hashedPwd with value stored in database
```

Example #1 Analysis

- Strengths:
  - Simple to implement
  - Original password cannot be easily recovered from hash
- Attacks:
  - Dictionary attack
  - Brute-force attack

Example #2: Adding Salt

- A salt is random data added to the password before it is hashed
- Salt is stored in the user table in unencrypted form

Username	Salt	Hashed value = hash('sha256', Salt + unencrypted_password)
fred	F2C2A523	72AE25495A7981C40622D49F9A52E4F1565C90F048F59
george	A1B2C324	91BC35495A7981C40622D49F9A52E4F1565C90F048F61

- The randomness and length of the salt are important
  - Use a cryptographic random number generator

Example #3: Cost Factor

- We want computing password hashes to take some time
  - Why?
- Password hash algorithms allow developer to specify a cost parameter that controls how much time the hash computation requires
  - Examples: Argon2, bcrypt
- OWASP suggests selecting a cost parameter that results in a 1 second computation time

Cryptographic Hash vs. Password Hash

Cryptographic Hash

- Fast
- Only one input (password)

Password Hash

- Intentionally slow
- At least three inputs:
  - Password
  - Per-user salt
  - Cost factor

PHP

- See https://secure.php.net/password_hash
- A single convenience function
  - Computes secure random hash
  - Combines with password and hashes result
  - Returns a value that incorporates algorithm, cost factor, and salt

Key Ideas

- Do not:
  - Store the password in a database/file in plaintext
  - Encrypt the password using a home-grown encryption system
  - Store the password using any encryption system that allows the original password to be recovered
- Do:
  - Store the password using an official encryption algorithm that does not allow the original password to be recovered
  - Make it difficult for hackers that acquire the encrypted passwords to be able to guess the originals using brute-force or dictionary attacks

Additional Reading

- OWASP Password Storage Cheat Sheet
  https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Password_Storage_Cheat_Sheet.md
- You Wouldn't Base64 a Password
  https://paragonie.com/blog/2015/08/you-wouldnt-base64-a-password-cryptography-decoded
- Libsodium: Cross-platform, cross-language library for password hashing and cryptography
  https://github.com/jedisct1/libsodium
- Safely Storing Passwords in several programming languages
  https://paragonie.com/blog/2016/02/how-safely-store-password-in-2016

Password Reset

Password Reset

- Most systems need a “Forgot Password” system
- How should it work?
  - Email a new random password to user, or require user to create his own new password?
  - Use security questions?
  - How should the flow work?

Password Reset Best Practice

- OWASP Forgot Password Cheat Sheet guidelines:
  1. Collect security questions when user registers initially.
  2. When user begins Forgot Password procedure, request answers to security questions.
  3. Do not generate a new password and send to user. Instead:
  4. Lock user's account and send a randomly generated code to side channel (email or SMS). Code should have a short expiration time frame.
  5. Allow user to change password within current session after entering randomly generated code.

Password Reset

- Further Reading
  - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Forgot_Password_Cheat_Sheet.md

Persistent Authentication (Remember Me)

Persistent Authentication (Remember Me)

- Problem:
  - After client authenticates, need to associate client with the authenticated account for a session that survives browser shutdowns

Non Solution: Session State

- Advantages:
  - Reasonably secure
- Disadvantages:
  - Session mechanisms are not very scalable
  - User must reauthenticate when session expires

Roll-Your-Own Solution

- Store an authentication token in a persistent cookie that associates browser with an authenticated user
- Security considerations:
  - Must prevent attacker from generating a token to impersonate an arbitrary user
    - Encrypt the authentication token
  - Attacker that obtains cookie can impersonate user until cookie expires
    - When user attempts to perform a sensitive operation, require reauthentication

Industry Solution: JSON Web Tokens

- A JSON web token is an authentication token in a standardized JSON format
- Token contains unencrypted data, cryptographically signed to ensure no tampering
- Can be used to associate a client with an authenticated user
- Supports a stateless authentication mechanism that allows a login session to persist securely across browser restarts
- JWT libraries available for the major web frameworks

JSON Web Tokens: How it works

- On authentication, server generates a signed token ("access token") that is associated with the authenticated user account
- Browser stores access token in persistent cookie or local storage
- Browser sends access token with each request
- Server uses access token to associate request with user account
- See examples/jwt

JWT Cookie Format

- Header + Payload + Data
- See jwt.io for details

JSON Web Tokens: Security Issues

- Problem: If attacker steals token, can impersonate user until token expires
  - Token should have short lifetime
- Problem: We don't want user to have to reauthenticate when token expires

JSON Web Tokens

- On authentication, server generates two encrypted tokens:
  - Short-lived access token
  - Long-lived refresh token
- Browser stores tokens in cookies or local storage
- Sends access token with each request
  - Server uses authentication token to associate request with user account
- When access token expires, server rejects request
  - Client code sends refresh request with refresh token
  - Server validates refresh token and generates new authentication token
- Example: https://www.geeksforgeeks.org/jwt-authentication-with-refresh-tokens/

Refresh Token Issues

- What if attacker steals refresh token?
  - Can impersonate user for a longer period
  - Access token is sent with every request; greater opportunity to steal
  - Refresh token is sent rarely; lower opportunity to steal
- Application should provide a way to invalidate tokens on user logout
  - Maintain a block list

Further Reading

- https://jwt.io/introduction
- https://docs.joshuatz.com/cheatsheets/security/jwt/
- https://www.digitalocean.com/community/tutorials/nodejs-jwt-expressjs