SUPERVVISOR: AHMED HANIF

NARWAL AUTH API DELIVERABLE 2

MUHAMMAD NASEEM
SAMI NAEEM
HASSAAN FAROOQ

Strengths

1) **Protection and Security:**
    - Server doesn't store password or hash, only public key Y
    - Each authentication attempt is unique
    - Protects against malicious servers misusing passwords.
    - Hard to attack due to randomized challenges
    - Immune to Man in the middle attacks
    - Protection against Phishing: As the actual password isn't transmitted, phishing attacks are less effective
    - Immunity to Password Database Breaches
    - Cross-Site Password Reuse Mitigation: Even if a user reuses passwords across sites, the unique public keys generated for each site protect against cross-site vulnerabilities.

2) **Scalability:**
    - Reduced Server-Side Computational Load, As the majority of the cryptographic computations are performed on the client-slide, hence reducing the server load,
    - No Need for Secure Password Storage Techniques, As the server doesn't need to implement complex password security strategies.
    - Facilitates Multi-Factor Authentication, As system can be easily extended to incorporate additional authentication factors.
    - Zero-Knowledge allows for easy auditing of the authentication process without exposing the sensitive information
    - Future-Proofing: The mathematical basis of the system (discrete logarithm problem) is believed to be resistant even to quantum computing attacks, providing some level of future-proofing.

3) **Policy Compliance:**
    - It compliance with privacy regulations, since it doesn't store the user passwords
    - Narwal auth API is in-line with the regulations of a lot – if not all – of the security and protection standards and policies.

Weaknesses:

JavaScript Dependency: The system requires the users to have JavaScript enabled in their browsers; about 2% of the users have it disabled. This limits its usability for those users.

Proposed Solution: We can use browser extensions or client-side scripts could be used as fallback options for users who have JavaScript disabled.

Key Management: The system relies on the server to maintain a consistent public key.

Proposed Solution: We can implement a proper robust key management system by including:

- Regular key rotation schedules: By changing the keys periodically, we can limit the amount of time a single key can be used, which reduces the potential impact if the key is compromised.
- Emergency Key Replacement: This will be a predefined process for quickly revoking (invalid). If the key is suspected to be compromised, it should be replaced with a new key.

Potential for Server Overload: The system could be vulnerable to high traffic flows, potentially crashing the server if there are too many brute force attempts to crack a user's password.

Proposed Solution: Restricting the number of authentication attempts from the single user i.e. 4-6 attempts during that specified time frame. Also, we can deploy a load-balancing technique to distribute the authentication requests across multiple servers, thus reducing the load on a single server.

Limited libraries for Cryptographic support: There are difficulties with JavaScript's support for cryptographic functions and large integer operations, forcing them to use workarounds

Proposed Solution: While we can use well-maintained javascript libraries like js-nacl or crypto-js, there's a need for better cryptographic libraries that need to be developed for working around certain cryptographic functions.

Session Hijacking: If HTTP is used, it is vulnerable to session hijacking attacks where an attacker steals the session ID of the user,

Proposed Solution: The use of HTTPS can prevent this type of vulnerability.

Susceptibility to DoS Attacks: complex cryptographic operations, such as hashing, modular exponentiation, and secure random number generation would exhaust the system's CPU and memory resources quickly making it vulnerable to DOS attacks. This underscores the critical need to address any vulnerabilities that could facilitate DoS attacks, ensuring the system's resilience and availability.

Proposed Solution: We should leverage DDoS protection services like Cloudflare or AWS Shield to absorb and mitigate large-scale attacks.

Support of Salting in public key: The inability to salt public keys weakens the security against precomputed attacks. Adding salt to public keys stored on the server could spoil this. while distributing the salts.

Proposed Solution: Implement a robust key to securely generate, store, and distribute salts for public key. Ensure that salts are unique for each user and updated

Complex locks slightly slow opener: Since it adds security, it also involves more complex steps based on mathematical patterns as compared to regular locks.This might slow down the login process

Solution: Optimize the mathematical algorithms and implement efficient computation techniques to reduce latency. Additionally, employ asynchronous processing to handle complex steps

Algorithm Weaknesses

Our Present Algorithms:

1) **Registration Algorithm:**
    - The client hashes the user's password to produce ‘x’.
    - The client computes ‘Y = g ^ x’ using the server's public key ‘g’.
    - The client sends the pair ‘(username, Y)’ to the server.
    - The server stores this information without ever seeing the user's password.
2) **Authentication Algorithm**:
    - The server generates and sends a random challenge ‘a’ to the client.
    - The client computes:
    - x = hash(password)
    - Y = g^x
    - r, a private random number
    - T = g^r
    - c = hash(Y || T || a)
    - z = r - c * x
    - The client sends the pair (c, z) to the server.
    - The server computes T' = Y^c * g^z and verifies if hash(Y || T' || a) == c.
    - If they match, the authentication is successful.

Problem: Precomputation Vulnerability

Solution: To remediate this vulnerability, we can introduce salting to the public key during registration and storing the pair as ‘(username, Y, s)’.

- During registration: ‘Y = g^(x + s)’
- During authentication:’ x = hash(password)’, ‘Y = g^(x + s)’, ‘T = g^r’, ‘c = hash(Y || T || a)’, ‘z = r - c * (x + s)’.

Our Enhanced Algorithms:

1) **Registration:**
    - Client computes ‘x = hash(password)’.
    - Client generates a random salt ‘s’.
    - Client computes ‘Y = g^(x + s)’.
    - Client sends ‘(username, Y, s)’ to the server.
    - Server stores ‘(username, Y, s)’.

2) **Authentication:**
    a) Server sends a random challenge 'a' to the client.
    b) Client computes:
        - x = hash(password)
        - Y = g^(x + s)
        - r, a private random number
        - T = g^r
        - c = hash(Y || T || a)
        - z = r - c * (x + s)
    c) Client sends (c, z) to the server.
    d) Server computes T' = Y^c * g^z.
    e) Server verifies if hash(Y || T' || a) == c.
    f) If they match, authentication is successful.