HTML and PHP Security

1.1 Authentication Technologies

HTTP Request-Based. HTTP Request-Based Authentication mechanisms directly embed user credentials within the HTTP request itself. While historically used, this approach is fundamentally insecure as it often exposes or weakly protects authentication data during transmission. Authentication in this manner is broadly handled through three major techniques:

(1) *Basic Authentication (HTTP Header)*: Credentials are sent within an HTTP header, typically the `Authorization` header, after being encoded using Base64. This is a simple reversible encoding scheme that translates binary data into a set of 64 standard ASCII characters (A-Z, a-z, 0-9, +, /, and = for padding). It is not encryption; its primary purpose is to ensure that binary data can be reliably transmitted over mediums that were originally designed for plain text, preventing data corruption.

(2) *NTLM (NT LAN Manager)*: This technique combines HTTP requests with the proprietary NTLM protocol. It is a challenge-response protocol that attempts to verify a user's identity without sending the password over the network.

(3) *Challenge-Response Mechanisms (MD5)*: These mechanisms use an MD5 hashing function to process a shared secret and a server-issued random value to generate a response. The server then compares this response to its own calculated hash. However, MD5 is cryptographically broken and is no longer considered secure for password hashing or challenge-response protocols due to vulnerability to collision and brute-force attacks.

Form-Based. Form-Based authentication is the most common method of web authentication, where the user supplies credentials via an interactive HTML form displayed on a webpage. The user inputs their username and password into the designed fields. Upon submission, the data is packaged into an HTTP request, typically a POST request. The credential parameters are sent within the request body. Alternatively, though highly insecure, they could theoretically be appended to the URL as query parameters.

The security of Form-Based authentication largely relies on how the transmission is protected. If the application uses HTTP instead of HTTPS, the entire request, including the credentials in the request body, is transmitted in plaintext and can be easily intercepted by an attacker performing a man-in-the-middle attack. This method is particularly susceptible to phishing attacks, where an attacker creates a fake, malicious login page that mimics the legitimate one to trick users into submitting their credentials.

Client SSL Certificates. Client SSL Certificates enforce authentication where both the client and the server must cryptographically prove their identities to each other before establishing a connection. This is a two-way authentication process that ensures a high level of security:

(1) *Server Proves Identity*: The client initiates the connection and requests a resource. The server responds by presenting its Server Certificates (which contains its public key). The client verifies this certificate to ensure it's communicating with the legitimate server.

(2) *Client Proves Identity*: If the server's identity is successfully verified, the server then requests the client to present its Client Certificate.

(3) *Validation and Access*: The server validates the client's certificate against a trusted list. If the certificate is valid, the client's identity is confirmed, and access is granted.

1.2 Authentication Design Flaws

The most significant flaw in any authentication system is the acceptance of weak passwords. These include blank passwords, credentials identical to the username, common dictionary words, or using factory default passwords. Implementing and enforcing a robust password policy is essential to mitigate these risks.

Brute-Force Attacks. Web applications must implement strong defenses against brute-force attacks, which systematically try to guess credentials by attempting numerous combinations. The flaw exploited here is the lack of a limit on the number of failed login attempts. Types of brute-force attacks include:

- *Simple Brute-Force*: Exhaustively trying every possible character combination within a defined length.
- *Dictionary Attack*: Using lists of common passwords, leaked passwords, or user-specific words.
- *Hybrid Attack*: Combining dictionary words with numerical or symbol permutations to increase complexity while remaining focused.
- *Rainbow Tables*: Pre-computed tables of password hashes used for offline cracking, particularly effective when an application stores weakly-hashed passwords.
- *Reverse Brute-Force*: Starting with a single, common password and trying it against a large list of usernames.

These tools are widely used by security professionals to test the resilience of authentication systems:

- *Burp Intruder*: A component of the Burp Suite, this tool automates customized attacks against web applications by repeatedly sending a base HTTP request with various payloads inserted into defined positions. It's highly flexible for brute-forcing, fuzzing for input-based vulnerabilities, and enumerating valid identifiers by analyzing differences in server responses (e.g., status code, length).
- *Turbo Intruder*: A Burt Suite extension designed for attacks requiring exceptional speed and volume. It utilizes a custom, high-speed HTTP stack and Python scripting for flexible attack configuration. It's ideal for high-concurrency brute-forcing and complex, multistep attack sequences like race conditions, sending vast numbers of requests in a short time.
- *WFUZZ*: A command-line tool known as a "Web Application Fuzzer". It is payload-driven; the user identifies where to inject data in an HTTP request (marked by the keyword `FUZZ`), and Wfuzz replaces that keyword with values from a user-defined wordlist.
- *THC-Hydra*: An open-source, fast network login hacking tool designed to perform dictionary attacks and brute-force attacks against numerous protocols and services (e.g., HTTP, FTP,

SSH, RDP, databases). It is notable for its versatility and parallelization capabilities, which allow it to launch multiple connection attempts simultaneously to rapidly test credentials.

From the web application's perspective, employing strong defense-in-depth strategies for credential handling is crucial.

Strong Credentials and Storage.

(1) *Strong Password Policy Enforcement*: Applications must enforce the use of strong passwords by avoiding short lengths, requiring a mixture of character types (uppercase, lowercase, digits, and special characters), actively checking against lists of common/dictionary passwords, and allowing for very long passwords (the longer, the better).

(2) *Secretive Handling*: Credentials must be handled securely in transit using SSL/HTTPS encryption to prevent eavesdropping.

(3) *Password Hashing*: Passwords are never stored in plaintext. They are processed by a cryptographic hashing function that generates a unique, fixed-length string corresponding to the original password. To enhance hash security, a salt is used. A salt is a unique, randomly generated value that is prepended or appended to the password before it is hashed. This ensures that even if two users choose the same password, the resulting hash in the database will be different, effectively preventing attackers from using Rainbow Tables and slowing down general brute-force attempts.

Authentication Flow Security.

(1) *Brute-Force Mitigation*: Applications must actively prevent brute-force attacks by:

- Rate Limiting: Slowing down subsequent login attempts after a few failures to make high-volume automated attacks impractical.
- Account Lockout: Implementing a strict limit on the number of consecutive failed logins before locking the account for a defined period.
- CAPTCHAs: Requiring a successful CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) after repeated failures to force a human check.

(2) *Username Enumeration Prevention*: Error messages should be generic (e.g., "Invalid username or password") to prevent an attacker from easily determining whether an account exists.

(3) *Validation Integrity*: Credentials must be validated properly; the application should never truncate a long password before hashing or validation, as this can undermine the security of complex passphrases.

(4) *Multi-Factor Authentication*: MFA must be implemented carefully. The user's primary credential (password) should be sent only once. All subsequent verification steps (e.g., handling the second factor code) should be managed server-side, ensuring all intermediate data is handled securely and statefully.

(5) *Password Change Policy*: Password changes should only be permitted for authenticated users to prevent unauthorized changes, and the change form should also be protected by measures like CAPTCHAs and session tickets.

These tools are utilized for offline password cracking against stolen password hashes.

- *Hashcat*: A powerful, open-source password recovery utility often considered the world's fastest by leveraging the parallel processing power of a computer's GPU. It supports various attack modes (e.g., Dictionary, Brute-Force, Hybrid, Mask) against a huge number of hashing algorithms.
- *John the Ripper*: A free and open-source command-line password cracking tool used for both security auditing and password recovery. JtR Can automatically detect the hash type and supports multiple cracking modes, including Dictionary Mode, Single Crack Mode, and Incremental Mode.
- *CrackStation*: This is not a standalone software tool but an online lookup service that performs a reverse hash lookup. Users submit a hash, and the service checks it against a massive, pre-computed database of billions of known plaintext passwords and their corresponding hashes. This allows for instantaneous cracking of weak, unsalted, or commonly used password whose hashes are already in the database.

Attacking Session Tokens. Session tokens are credentials, typically long, pseudo-random strings embedded in cookies or HTTP headers, sent by the server to the client to enforce a persistent, trusted session after initial authentication. If compromised or predictable, these tokens can be exploited to establish a forged session with the server, bypassing the login process entirely. Token vulnerabilities arise when the token generation logic is flawed, making the tokens guessable or reversible.

(1) *Meaningful Tokens*: These tokens directly encode user-specific information (e.g., user ID, role, or expiry time). They often use basic, reversible encoding schemes like Hexadecimal or Base64. Once the encoding is identified, an attacker can decode a valid token, modify the plaintext data, and then re-encode the string to create a forged token. This is often much more efficient than brute-forcing credentials.

(2) *Predictable Tokens*: These tokens do not necessarily embed user data but follow a discernable, non-random logic for their generation. Tokens being generated sequentially, tokens being structured predictably, reliance on the generation time, or the use of a Weak Random Number Generator, which makes the next token in the sequence easy to guess.

(3) *Encrypted Tokens*: These tokens encrypt user information using symmetric encryption algorithms (like DES, AES, or others) to protect the integrity of the data. Attackers can exploit weaknesses in the chosen block cipher mode, such as Electronic Codebook (ECB) mode or Cipher Block Chaining (CBC) mode, allowing for analysis or manipulation of the encrypted blocks without knowing the key.

JSON Web Token (JWT) is an open standard defining a compact and self-contained format for securely transmitting information as a JSON object. This information is digitally signed for integrity and trust. A JWT consists of three parts separated by dots (`.`):

(1) *Header*: Contains the metadata about the token, including the signature algorithm used (e.g., HMAC with SHA-256 or RSA).

(2) *Payload*: Contains the claims about an entity and additional data. These claims are often viewable but must be trusted only because they are signed.

(3) *Signature*: Used to verify the sender of the JWT and ensure the token hasn't been tampered with. The signature is created by taking the Base64-encoded header, the Base64-encoded payload, a secret key, and running it through the specified signing algorithm.

The security of a JWT relies entirely on the secrecy of the signing key and the strength of the algorithm. If the secret key is leaked, an attacker can forge a valid signature, creating a malicious token that the server will trust.

1.3 Lab 3

Most of this assignment's tasks will concern attacking the Damn Vulnerable Web Application (DVWA). You will be asked to solve tasks by changing their difficulty progressively. To change it, go to DVWA and select the difficulty by clicking on DVWA settings

Task 1 - Dictionary-Based Brute-Force

Attack the Brute-Force section of the DVWA at easy and medium levels by performing a dictionary attack. For each level, you are required to retrieve the passwords for the users: `pablo` , `admin` , `1337` , `gordonb` , and `smithy` . For good password sets, check: `https://github.com/danielmiessler/SecLists/tree/master/Passwords/Common-Credentials`