LULEÅ UNIVERSITY OF TECHNOLOGY

Lab 3: Authentication Security: From Password Storage to FIDO2/WebAuthn

Computer Security D7076E

Group 35

Vatousiadi Spyridoula – spyvat-5@student.ltu.se
Stefanos Ntentopoulos – stente-5@student.ltu.se

October 19, 2025

1 Introduction And Objectives

This laboratory assignment focuses on implementing and analyzing secure authentication mechanisms, from basic password storage to advanced multi-factor authentication systems. The work encompasses fundamental password security, API development, cryptographic attacks, multi-factor authentication, FIDO2/WebAuthn implementation, and sophisticated attack demonstrations.

The assignment aims to provide hands-on experience with modern authentication security challenges, demonstrating both vulnerabilities and their mitigation. Through practical implementation and attack simulations, we explore the evolution from simple password-based authentication to robust multi-factor systems that resist contemporary threats. In general, we implement and compare four password hashing algorithms with different security characteristics, a secure authentication REST API, a secure authentication REST API with integrated MFA support, security testing, and three distinct MFA methods.

Also, we demonstrate some attacks like password cracking, timing attacks, and MITM relay attacks, and implement various security mitigations.

2 Password Hashing

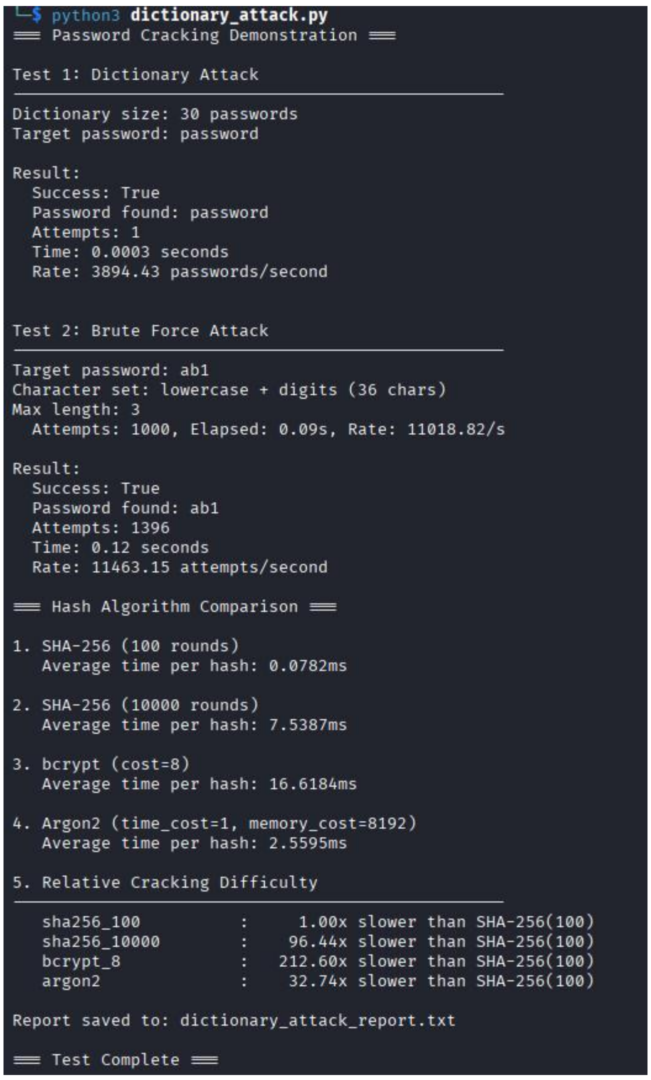
With the Flask application (app.py) we implement a secure authentication REST API, which initializes a SQLite database to store user credentials, which are hashed with four different hashing algorithms/methods. These are the SHA-256, SHA-3, bcrypt, and Argon2.

The SHA-256 (Secure Hash Algorithm 256-bit) belongs to the SHA-2 family, designed by the NSA and published in 2001. It processes data in 512-bit blocks and produces a 256-bit hash value. It provides fast computation, it is well-studied and standardized, it does not have any known practical collisions, and it is available in all programming languages and is deterministic and reproducible. On the other hand, it is too fast for password hashing, making it vulnerable to rapid brute-force, it is also vulnerable to GPU/ASIC acceleration and is vulnerable to rainbow table attacks without salting.

As we can see in the following figure 1, the cracking rate is almost 11,463 attempts/second.

The SHA-3, is based on the Keccak algorithm, it has different internal structure from SHA-2 and is more resistant to length-extension attacks with flexible output length, but it still has the similar speed to SHA-256, still being too fast for passwords, it is less widespread and is vulnerable to same GPU/ASIC acceleration.

So both SHA-256 and SHA-3, are cryptographically secure general-purpose hash functions but inadequate for password storage without additional measures. Their speed which was designed as a feature for general hashing, becomes a vulnerability in password context.



```
└─$ python3 dictionary_attack.py
══ Password Cracking Demonstration ══

Test 1: Dictionary Attack
──────────────────────────────────────────────────
Dictionary size: 30 passwords
Target password: password

Result:
  Success: True
  Password found: password
  Attempts: 1
  Time: 0.0003 seconds
  Rate: 3894.43 passwords/second


Test 2: Brute Force Attack
──────────────────────────────────────────────────
Target password: ab1
Character set: lowercase + digits (36 chars)
Max length: 3
  Attempts: 1000, Elapsed: 0.09s, Rate: 11018.82/s

Result:
  Success: True
  Password found: ab1
  Attempts: 1396
  Time: 0.12 seconds
  Rate: 11463.15 attempts/second

══ Hash Algorithm Comparison ══

1. SHA-256 (100 rounds)
   Average time per hash: 0.0782ms

2. SHA-256 (10000 rounds)
   Average time per hash: 7.5387ms

3. bcrypt (cost=8)
   Average time per hash: 16.6184ms

4. Argon2 (time_cost=1, memory_cost=8192)
   Average time per hash: 2.5595ms

5. Relative Cracking Difficulty
──────────────────────────────────────────────────
   sha256_100          :     1.00x slower than SHA-256(100)
   sha256_10000        :    96.44x slower than SHA-256(100)
   bcrypt_8            :   212.60x slower than SHA-256(100)
   argon2              :    32.74x slower than SHA-256(100)

Report saved to: dictionary_attack_report.txt

══ Test Complete ══
```

The bcrypt is a password hashing function based on the Blowfish block cipher. Its key innovation is the adjustable work factor (cost parameter), which controls the computational effort required to hash a password. This makes bcrypt deliberately slow, helping to thwart brute-force attacks. It was created by Niels Provos and David Mazières in 1999. When we call the function bcrypt(password, salt, cost), it combines the password and salt, then uses them to create the Blowfish key setup. bcrypt repeats the Blowfish key setup phase a number of times equal to two multiplied by itself as many times as the cost value, making the final hash increasingly complex.

Finally, Argon2 is a modern, memory-hard password hashing algorithm that emerged as the winner of the Password Hashing Competition (PHC) in 2015. Its design specifically addresses the weaknesses of earlier algorithms like bcrypt, particularly in the face of rapidly advancing hardware such as GPUs and ASICs. By making efficient password cracking economically and technically challenging, Argon2 sets a new standard for secure password storage. It achieves its security through memory-hardness, meaning it requires a significant amount of RAM to compute each hash. This property makes parallel attacks using specialized hardware much less effective, as attackers would need to invest in large amounts of memory for each cracking attempt, dramatically increasing the cost and complexity of large-scale attacks.

In our implementation, at app.py file

```
def hash_argon2(self, password):
    ph = Argon2PasswordHasher(time_cost=1, memory_cost=8192)
    peppered_password = (password + PEPPER.decode('utf-8', errors='ignore')).encode('utf
    return ph.hash(peppered_password)
```

A "pepper" is a secret value stored separately from the database, usually in environment variables or an external secure file and unlike the salt, which is unique per password and stored with it, the pepper is global and secret. It is added to the password for extra security. The Argon2PasswordHasher is configured with a time cost of 1 and a memory cost of 8192 KB, which can be adjusted for higher security.

Based on the comprehensive implementation and analysis conducted in this assignment, several critical security recommendations emerge for modern authentication systems. Argon2 should be the preferred choice for new systems as it provides the strongest security against modern GPU and ASIC-based attacks through its memory-hard design. For legacy systems requiring compatibility, bcrypt is better choice.Finally, we should never use fast, general-purpose hash functions for password storage like plain SHA-256 or SHA-3, MD5 which has been broken and is vulnerable to collisions or the SHA-1 (which is deprecated and is also vulnerable to collisions. The implementation of a salt and pepper architecture creates a defense-in-depth approach where even database compromise does not immediately expose user passwords.

3 MULTI-FACTOR AUTHENTICATION (MFA)

3.1 MFA Fundamentals

Multi-Factor Authentication (MFA) is a security system that requires a user to provide two or more distinct forms of verification from separate categories to prove their identity.

The three primary factors of authentication are:

Knowledge Factor (Something you KNOW): This is a secret known only to the user, such as a password, a Personal Identification Number (PIN), or the answer to a security question.

Possession Factor (Something you HAVE): This is a physical object in the user's possession, such as a smartphone (receiving an SMS or push notification), a hardware token, or a smart card.

Inherence Factor (Something you ARE): This is a biometric identifier unique to the user, such as a fingerprint, facial recognition, iris scan, or voice pattern.

Security Principle: The fundamental strength of MFA lies in the requirement for independent factors. A compromise of any single factor (e.g., a stolen password) is insufficient to gain access, as the attacker would also need to compromise the second factor (e.g., steal the user's phone).

3.2 TOTP (Time-based One-Time Password)

TOTP is an open standard (RFC 6238) that generates one-time passwords that are valid for a short period, typically 30 seconds. It is a time-based variant of the HMAC-based One-Time Password (HOTP) algorithm. The password is derived from a shared secret key and the current time.

The process goes as following: The server generates a cryptographically random secret key,then it presents this key to the user encoded within a QR code. The user scans the QR code with an authenticator app, which stores the secret key, making it a shared secret between the server and the user's device.

When the user attempts to log in with their username and password, the server prompts for a TOTP code, then the user opens their authenticator app, which computes the TOTP code using the stored secret key and the current time. The next step is for the user to enter the generated 6-digit code into the login form, while the server independently computes the expected TOTP code using its stored copy of the secret key and its own clock. Finally, the server verifies if the user-submitted code matches the server-generated code and allows the user to log in, if the username and the password are correct.

3.3 HOTP (HMAC-based One-Time Password)

HOTP is an open standard (RFC 4226) that generates one-time passwords based on a counter value rather than time. Each time a code is generated and used, the counter increments on both the client and server.

The main difference between TOTP and HOTP is the synchronization mechanism. TOTP uses time for automatic synchronization, while HOTP uses a counter that requires manual synchronization after each use.

A big problem with the HOTP is the counter de-synchronization problem. De-synchronization occurs when the client's counter and the server's counter fall out of sync. For example, a user generates several codes (e.g., counters 100, 101, 102) without submitting them, perhaps by accidentally opening the app. When they finally use code 103, the server is still expecting code 100. The verification fails. The solution that has been found is the look-ahead window, with this, the server checks not only its current counter value but also the next N values (e.g., 100 to 110). If a match is found at counter 103, the server accepts the code and updates its counter to 104. While a larger look-ahead window improves usability but slightly reduces security, as it gives an attacker a wider range of valid codes to guess from.

3.4 FIDO2/WebAuthn

FIDO2 instead of using shared secrets, it utilizes public key cryptography, as a result, it offers phishing-resistant security. It is based on two core standards, the WebAuthn (Web Authentication)which is a W3C standard that defines a browser API for creating and using public key-based credentials, and the CTAP (Client to Authenticator Protocol) that allows external authenticators to work with browsers.

The process for FIDO2/WebAuthn is as follows: The user visits the https://example.com and initiates registration, the server generates a random challenge. The browser calls navigator.credentials.create(), and after the user consents, the authenticator generates a new public-private key pair unique to example.com. The private key is stored securely on the authenticator and never leaves. While, the public key, along with a signed package that includes the challenge and a critical origin (the website's domain), is sent to the server. Finally, the server stores the public key and credential ID linked to the user's account. When the user tries to authenticate and login with the https://example.com, the server generates a new random challenge. The browser calls navigator.credentials.get(), and after the user consents to authenticate, the authenticator signs the challenge and the origin with the private key for example.com, the signed assertion is sent to the server, and after the server verifies the signature using the stored public key and also verifies that the origin in the signed data matches https://example.com, the server allows the user to log in.

3.5 MFA Comparison And Recommendation

As we mentioned, HOTP and TOTP are both based in symmetric cryptography, relying on a shared secret pre-established between the user's device and the authentication server. The distinction between them lies in their synchronization mechanism. HOTP is driven by an incrementing counter, each use of a code advances the counter, requiring the server to maintain synchronization, often through a look-ahead window to account for instances where a user generates codes without submitting them. This counter-based approach makes HOTP resilient to clock drift and entirely operable offline, but it introduces a significant usability challenge in the form of counter de-synchronization, which can frustrate users and requires complex server-side logic to resolve.

In contrast, TOTP replaces the counter with a time-based value, typically using 30-second intervals. This provides automatic synchronization, as both the client and server derive the same code from their independent clocks. This makes TOTP more user-friendly, as codes refresh automatically without any user action. However, this introduces a dependency on time synchronization, where significant clock drift between the device and server can lead to authentication failures, a problem mitigated by checking adjacent time windows.

Both HOTP and TOTP share a critical vulnerability, they are susceptible to real-time phishing and man-in-the-middle (MITM) attacks. Because the OTP code is a shared secret that the user can manually transcribe, an attacker can capture it on a fake login site and immediately relay it to the genuine service, thereby gaining access. The user has no built-in mechanism to verify the authenticity of the site requesting the code. This limitation means that while OTPs protect against credential stuffing and offline attacks, they do not provide phishing resistance.

FIDO2/WebAuthn is based in the asymmetric public-key cryptography. During registration, the user's authenticator generates a unique public-private key pair for the specific website. The private key never leaves the authenticator. The security feature, known as origin binding, cryptographically ties the key pair to the website's domain name. During authentication, the server sends a challenge, which the authenticator signs with the private key. The signed response includes the domain name of the site that initiated the request. The server then verifies both the cryptographic signature and that the domain in the signature matches its own. This process makes FIDO2 resistant to phishing. If a user is tricked into authenticating on a malicious site, the signed response will contain the attacker's domain, and the legitimate server will reject it immediately.

In terms of user experience, HOTP is often considered the least seamless, frequently requiring a button press to generate a code. TOTP offers a smoother experience with automatically refreshing codes. However, FIDO2 can provide the most streamlined interaction, often reducing the authentication step to a single biometric scan or touch, which is both more secure and more convenient.

In terms of implementation perspective, HOTP and TOTP are relatively simple to integrate using standardized libraries, whereas FIDO2 requires a more complex setup and require a secure HTTPS context.

In conclusion,while HOTP and TOTP operate within a model that is inherently vulnerable to phishing, TOTP is generally preferred over HOTP due to its automatic time-based synchronization, and low to no cost. But FIDO2/WebAuthn,is not only offering stronger security but also a better user experience, making it the recommended choice. In deployment the best recommendation is use first of all, FIDO2/WebAuthn with TOTP, which provides the best phishing resistance and we have the OTP as a fallback if something goes wrong. Then, only TOTP which takes advantage of the wide compatibility it offers, and it has no cost. Afterwards, only HOTP provided that it will be implemented only with specific hardware tokens. The choice of single-factor authentication without MFA is the worst case to choose nowadays.

4 The experiment setup

The test environment utilized a standard development machine running the Linux operating system, specifically Kali Linux. Python version 3.13.7 was used for all scripting and application development. The database system was SQLite3, and the web framework employed was the latest version of Flask. Key dependencies included the flask web framework, bcrypt for password hashing, argon2-cffi for Argon2 hashing, pyotp for TOTP and HOTP implementation, qrcode and pillow for QR code generation, fido2 for WebAuthn and FIDO2 support, and requests for HTTP client testing.

Password cracking tests involved dictionary attacks (dictionary_attack.py) using thirty common passwords and brute-force attacks targeting lowercase letters and digits for passwords of length one to four characters.

The hashing algorithms tested were SHA-256, SHA-3, bcrypt, and Argon2. The primary metrics recorded were attempts per second, time-to-crack, and success rate. (app.py, crack_passwords.py)

Timing attack tests compared naive string comparison against constant-time comparison. These tests used string lengths of eight, sixteen, and thirty-two characters, with ten thousand iterations per test case. Time was measured using time.time() with nanosecond precision, and results were analyzed for mean and standard deviation. (timing_attack.py)

Multi-factor authentication tests evaluated TOTP using valid codes, expired codes, and time windows of plus-or-minus zero and one. HOTP tests involved sequential counters and desynchronization scenarios. WebAuthn tests covered registration, authentication, and origin mismatch cases. (mfa_totp.py, mfa_hotp.py, fid02_webauthn.py)

Man-in-the-middle relay tests were conducted between a real server at http://localhost:5000 and a MITM proxy at http://localhost:8080, using a test user with TOTP enrolled (integrated_app.py, mitm_proxy.py)

All tests were executed manually execution and curl commands for API testing were also performed, alongside interactive demonstrations. Results were logged to JSON files , text files with the suffixes _results.txt and _report.txt, and various screenshot files in JPG and PNG formats.

5 Conclusion

The implementation of this lab assignment demonstrates that password security depends not only on the choice of hashing algorithm but also on enforcing sufficient password length and educating users about strong password practices. The use of salt and pepper significantly enhances protection against common attacks, though proper management is essential. Timing attacks can be effectively mitigated by adopting constant-time comparison methods, with minimal performance trade-offs. Among multi-factor authentication methods, WebAuthn stands out for its strong resistance to phishing and relay attacks, making it the preferred choice for high-security applications, while TOTP remains a practical fallback for broader compatibility. Overall, a layered approach combining slow hash functions, proper use of salt and pepper, constant-time operations, and modern MFA solutions like WebAuthn provides the highest level of security for authentication systems.

6 References

1. RFC 6238 - TOTP: Time-Based One-Time Password Algorithm `https://datatracker.ietf.org/doc/html/rfc6238`
2. RFC 4226 - HOTP: An HMAC-Based One-Time Password Algorithm `https://datatracker.ietf.org/doc/html/rfc4226`
3. RFC 2104 - HMAC: Keyed-Hashing for Message Authentication `https://datatracker.ietf.org/doc/html/rfc2104`
4. W3C WebAuthn Specification `https://www.w3.org/TR/webauthn-2/`
5. FIDO2/CTAP2 Specifications `https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html`
6. Flask Documentation `https://flask.palletsprojects.com/`
7. pyotp Documentation `https://pyauth.github.io/pyotp/`
8. python-fido2 Documentation `https://github.com/Yubico/python-fido2`
9. Argon2 Documentation `https://argon2-cffi.readthedocs.io/`
10. OWASP Password Storage Cheat Sheet `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`
11. OWASP Authentication Cheat Sheet `https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html`
12. Computer Security: Principles and Practice, 4th Global Edition, authored by William Stallings and Lawrie Brown
13. Geeks For Geeks `https://www.geeksforgeeks.org/`