

Assignment 2: Cracking Hashes and Exploring Diffie-Hellman Key Exchange Vulnerabilities

Shivansh Garg 2024MCS2450 , Rohit Patidar 2024JCS2042

January 28, 2025

Subject: SIL 765 Network and System Security

Introduction

The assignment is divided into two parts, each focusing on exploring specific vulnerabilities in cryptographic algorithms. The objective of the first part of the assignment is to understand the vulnerabilities of hashing algorithms by performing attacks on both salted and unsalted hashes. This involves writing Python programs to crack hashes generated using MD5, SHA1, and SHA256 algorithms, performing the attacks on both salted and unsalted hashes, and analyzing the effectiveness and difficulty of these attacks. A comparison of the results for salted versus unsalted hashes is also conducted.

The second part of the assignment aims to explore vulnerabilities in the Diffie-Hellman key exchange mechanism by simulating a secure communication scenario. In this part, we interact with a server (oracle) that implements a vulnerable version of the Diffie-Hellman protocol and attempt to uncover the private key used by the server through analysis of the protocol's weaknesses.

Part-1 Cracking Hashes with and without Salt

Process

Task-1 : Cracking Unsalted Hashes

The cryptanalysis process involves the following steps:

1. **Reading Files:** A list of plain-text passwords is read from a file (e.g., `rockyou.txt`), and hash lists (MD5, SHA1, SHA256) are loaded from separate files.
2. **Hash Matching:**
 - For each hashing algorithm, the program computes the hash of every password.

- The computed hash is compared with the preloaded hash list.
 - If a match is found, the hash and corresponding password are stored.
3. **Time Measurement:** The total time taken for matching is recorded for each algorithm.
 4. **Output:** Matched hashes and passwords are saved in an output file, and a summary of matches for each hash type is displayed.

This process evaluates the time efficiency of brute-forcing hashes and validates the strength of password security.

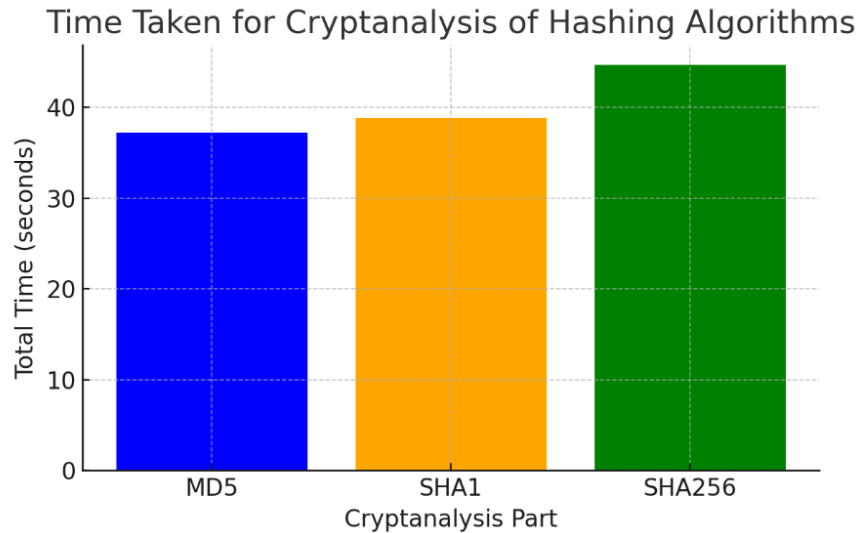


Figure 1: Time Taken VS Model

Task-2 : Cracking salted Hashes

This script performs a cryptanalysis attack on password hashes using MD5, SHA1, and SHA256 algorithms. The process can be broken down as follows:

1. **Reading Passwords and Hashes:** It reads common passwords from the 'rockyou.txt' file and salted hashes from three separate files for MD5, SHA1, and SHA256.
2. **Salted Hashes:** The salted hash consists of a salt and the hash. The script extracts the salt and hash separately.
3. **Hashing and Matching:** For each salt, the script applies the corresponding hashing algorithm (MD5, SHA1, SHA256) to each password concatenated with the salt, then compares it to the hash.

4. **Storing Matches:** If a match is found, the password is stored in a dictionary, and the results are written to an output file.
5. **Performance:** The script tracks the time taken for each hashing algorithm (MD5, SHA1, SHA256).

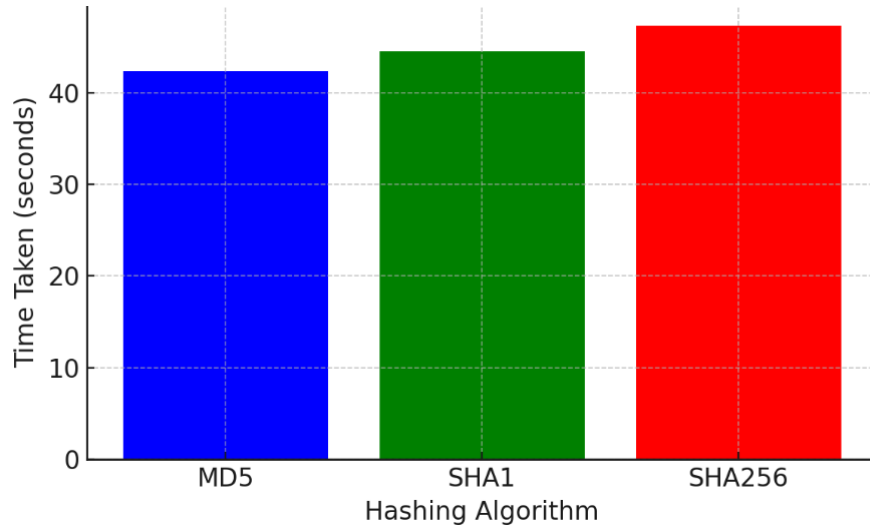


Figure 2: Time Taken VS Model

Challenges and Limitations in Hash Cracking

1. Challenges of Cracking Salted Hashes

The provided code demonstrates cracking unsalted hashes (MD5, SHA1, SHA256) by comparing plaintext passwords to stored hashes. Salting, which appends a unique random value to each password before hashing, complicates this process by:

- Making precomputed attacks like rainbow tables ineffective, as each hash is unique. Without salts, identical passwords produce identical hashes, allowing attackers to identify and target common passwords across multiple accounts. Salts ensure that even if two users have the same password, their hashes differ, making it infeasible to exploit reused passwords.
- Requiring the attacker to recompute hashes for every combination of passwords and salts.
- Salts prevent attackers from leveraging parallel processing techniques effectively, as each hash requires individualized cracking attempts tied to its unique salt.

- The introduction of a unique salt per password means that attackers must perform a separate brute-force attack for each hashed password, exponentially increasing the computational resources and time required to crack multiple hashes

The code highlights the ease of cracking unsalted hashes, emphasizing the importance of salting for enhanced security.

2. Security Enhancement Through Salts

- They guarantee that each hash is unique, even for identical passwords, thereby preventing attackers from identifying patterns or reusing their efforts across multiple accounts.
- By adding randomness, salts make it impossible to predict the hash output without knowing the exact salt value, thereby enhancing the unpredictability of the hash.
- When combined with strong hash functions, salts contribute to a layered security approach, making it significantly harder for attackers to reverse-engineer the original input.

3. Why MD5 and SHA1 Are Insecure

The code uses MD5 and SHA1 for hash matching, but these algorithms are outdated:

- **MD5:** Vulnerable to collision attacks, where two inputs produce the same hash.
- **SHA1:** Though stronger than MD5, it is also prone to collisions due to advances in computational power.
- **Speed and Efficiency :**Both MD5 and SHA1 are designed to be fast, which, while beneficial for performance, is detrimental for security in hashing passwords. Their speed allows attackers to perform rapid brute-force and dictionary attacks, increasing the likelihood of successfully cracking hashes.

The inclusion of SHA-256 in the code showcases its advantages, such as longer hash lengths and stronger collision resistance, making it more secure.

4. Advantages of SHA-256 :

- Part of the SHA-2 family, SHA-256 offers a significantly higher level of collision resistance compared to SHA1 and MD5, making it computationally infeasible to find two distinct inputs that produce the same hash.

- SHA-256 offer longer hash outputs (256 bits and beyond), which increases the complexity for attackers attempting to perform brute-force attacks, thereby enhancing security.
- While being more computationally intensive than MD5 and SHA1, the increased computation time is advantageous for security, particularly in password hashing where it slows down attackers attempting to crack hashes.
- SHA-256 provide robust defenses against both preimage (finding an input that hashes to a specific output) and second preimage (finding a different input with the same hash) attacks, ensuring the integrity and uniqueness of the hashes
- SHA-256 is widely recognized and standardized by security organizations, ensuring ongoing support, updates, and scrutiny by the cryptographic community, which further solidifies their security posture.

5. Limitations of Relying Solely on Hash Cracking

The code iterates over a wordlist to match hashes, but practical systems employ protections that limit the effectiveness of such attacks:

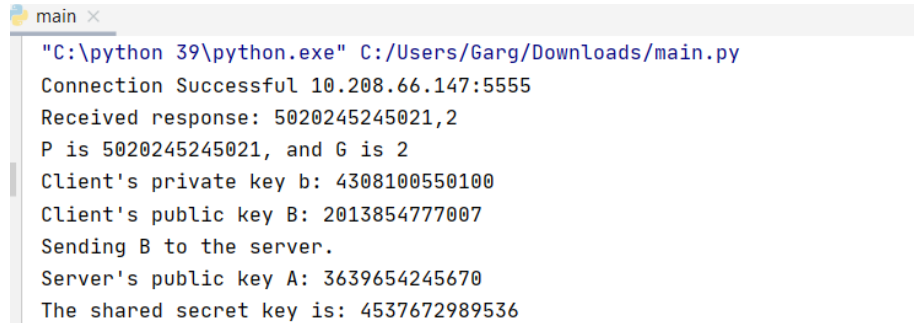
- **Rate-Limiting:** Systems restrict repeated login attempts, slowing brute-force attacks.
- **Hash Iterations:** Modern algorithms like bcrypt and Argon2 use multiple iterations, increasing computation time for each guess.
- **Salts and Key Derivation Functions:** These add computational complexity, making attacks infeasible within reasonable timeframes.

Part-2 Exploring Diffie-Hellman Key Exchange Vulnerabilities

Task - 1,2

- **Client-Server Connection:** The client sends an entry number to the server. The server responds with a generator G and a large prime number P .
- **Client-side Key Generation:** The client generates a private key k_a and computes the public key $K_a = G^{k_a} \mod P$.
- **Server-side Key Generation:** The server generates a private key k_b and computes the public key $K_b = G^{k_b} \mod P$.
- **Key Exchange:** The client sends its public key K_a to the server, and the server sends its public key K_b to the client.

- **Shared Secret Key Generation:** The client computes $S = K_b^{k_a} \mod P$, while the server computes $S = K_a^{k_b} \mod P$.
- **Result:** Both the client and the server now have the same shared secret key S for secure communication.



```

main x
"C:\python 39\python.exe" C:/Users/Garg/Downloads/main.py
Connection Successful 10.208.66.147:5555
Received response: 5020245245021,2
P is 5020245245021, and G is 2
Client's private key b: 4308100550100
Client's public key B: 2013854777007
Sending B to the server.
Server's public key A: 3639654245670
The shared secret key is: 4537672989536

```

Figure 3: Connection Establishment

Task - 3

Brute Force Attack

- **Brute Force Attack:** To determine the server's private key a , the client performs a brute force attack. The client iterates through all possible values of the private key (from 1 to $P - 1$) and checks if $G^i \mod P = A$. Once the correct private key a is found, the client has successfully guessed the server's private key.
- **End of Attack:** The brute force attack terminates once the correct private key of the server is found.
- **Result:** When the Entry number was 2024MCS2450 then the server private key was 99, and when the entry number was 2024JCS2442 then the server's private key was 54.

```
main x
"C:\python 39\python.exe" C:/Users/Garg/Downloads/main.py
Connection Successful 10.208.66.147:5555
Received response: 5020245245021,2
P is 5020245245021, and G is 2
Client's private key b: 4499979667328
Client's public key B: 4564374943023
Sending B to the server.
Server's public key A: 3639654245670
The shared secret key is: 1900396234292
Started guessing server's private key
The private key of server is 99
Connection closed.
```

Figure 4: Brute-Force Attack

```
main x
"C:\python 39\python.exe" C:/Users/Garg/Downloads/main.py
Connection Successful 10.208.66.147:5555
Received response: 4220244244217,3
P is 4220244244217, and G is 3
Client's private key b: 3734837343655
Client's public key B: 2202376546635
Sending B to the server.
Server's public key A: 729547676528
The shared secret key is: 1196923970747
Started guessing server's private key
The private key of server is 54
Connection closed.
```

Figure 5: Brute Force with different Entry Number

Man in the Middle attack

Process-1

The Man-in-the-Middle (MITM) attack in this part is implemented on a Diffie-Hellman key exchange protocol, where the attacker intercepts and modifies the communication between a client and a server (Oracle). The steps are as follows:

1. **Intercept Initial Connection:** The attacker listens for incoming client connections and establishes a connection with the Oracle server. This enables the attacker to act as a relay between the client and Oracle.

2. **Forward Parameters (P, G):** The client sends an entry number to the attacker, which is forwarded to the Oracle. The Oracle responds with the parameters P (a prime number) and G (a generator modulo P). These are intercepted by the attacker and then forwarded to the client.
3. **Intercept Public Key Exchange:**
 - The client sends its public key $B = G^b \mod P$ (where b is the client's private key) to the attacker.
 - Instead of forwarding B to the Oracle, the attacker generates their own private key c and computes $C = G^c \mod P$. This C is sent to both the Oracle and the client as the other party's public key.
4. **Establish Separate Shared Secrets:**
 - The Oracle computes its shared secret with C as $S_{\text{server}} = C^a \mod P$ (where a is the Oracle's private key).
 - The attacker computes their shared secret with the client as $S_{\text{client}} = B^c \mod P$ and with the Oracle as $S_{\text{server}} = A^c \mod P$ (where A is the Oracle's public key).
5. **Result:** The attacker now has separate shared secrets with both the client and the Oracle. This allows the attacker to decrypt and modify messages between the two parties, effectively breaking the confidentiality of the communication.

```
"C:\python 39\python.exe" C:/Users/Garg/Downloads/main.py
Connection Successful 10.194.53.144:6666
Received response: 5020245245021,2
P is 5020245245021, and G is 2
Client's private key b: 1621993291850
Client's public key B: 3904897996768
Sending B to the server.
Server's public key A: 1073741824
The shared secret key is: 3211671722836
Started guessing server's private key
The private key of server is 30
Connection closed.
```

Figure 6: Victim Side


```

rohitpatidar@Rohits-MacBook-Air Part_2 % python3 attacker.py
[MITM] Listening on 0.0.0.0:6666...
[MITM] Accepted connection from ('10.194.48.207', 55736)
[MITM] Connected to Oracle Server at 10.208.66.147:5555
[MITM] Received entry number from victim: 2024MCS2450
[MITM] Forwarded entry number to Oracle: 2024MCS2450
[MITM] Received (P, G) from Oracle: 5020245245021,2
[MITM] Forwarded (P, G) to victim.
[MITM] Parsed P: 5020245245021, G: 2
[MITM] Received victim's public key (B): 3904897996768
[MITM] Chose own private key (c): 30
[MITM] Computed public key (C): 1073741824
[MITM] Sent C to Oracle as victim's public key.
[MITM] Sent C to victim as Oracle's public key.
[MITM] Received server's public key (A): 3639654245670
[MITM] Computed shared secret with server (S_server): 4691737358632
[MITM] Computed shared secret with victim (S_victim): 3211671722836
[MITM] Private key of Oracle's Server : 99
[MITM] Closed connection with victim.
[MITM] Closed connection with Oracle.
[MITM] Connection handling complete.
rohitpatidar@Rohits-MacBook-Air Part_2 %

```

Figure 7: Attacker Side

Process-2

Attack Model and Assumptions

The goal of this experiment was to conduct ARP Spoofing on a mobile hotspot network using **Ettercap** to intercept traffic between devices. The assumption was that devices connected to the hotspot would rely on ARP for MAC-IP mapping, allowing an attacker to poison the ARP cache and redirect traffic.

Setup and Tools

The tools and setup used for the experiment were as follows:

- **Attacker's device:** A Linux machine running **Ettercap**.
- **Victims:** Multiple devices connected to the mobile hotspot.

Process

The steps followed in the experiment were:

1. Connected the attacker and victim devices to the mobile hotspot.
2. Launched **Ettercap** in graphical mode to scan and identify devices:

```
sudo ettercap -G
```

3. Started ARP poisoning using **Ettercap** to associate the gateway's IP with the attacker's MAC address.

Results

The experiment was unsuccessful:

- Ettercap could not detect other devices on the network.
- ARP poisoning failed as the mobile hotspot appeared to isolate the devices, preventing communication between them.
- The ARP cache was not poisoned successfully.

Challenges

The main challenges encountered during the experiment were:

- **Client Isolation:** The mobile hotspot enforced client isolation, preventing devices from communicating directly with each other.
- **Limited Control:** Mobile hotspots typically lack advanced configuration options, such as disabling client isolation.

Conclusion

The experiment was hindered by the mobile hotspot's client isolation feature. ARP Spoofing requires devices to communicate with each other, which was blocked in this setup. For successful ARP Spoofing, a more configurable router or network without client isolation is recommended.

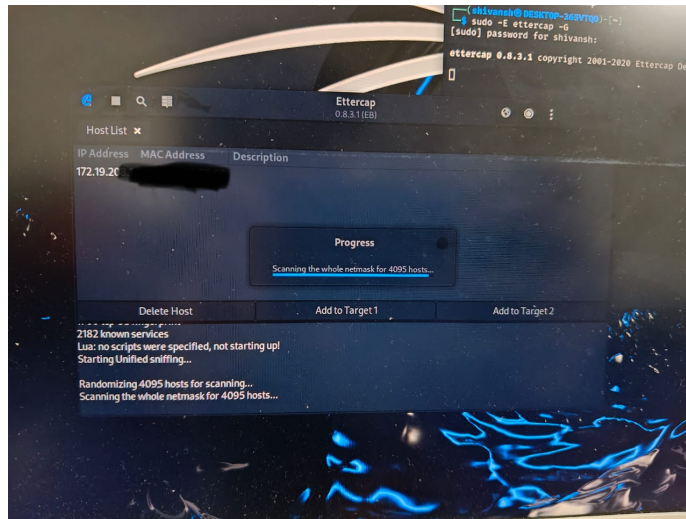


Figure 8: Ettercap finding the devices

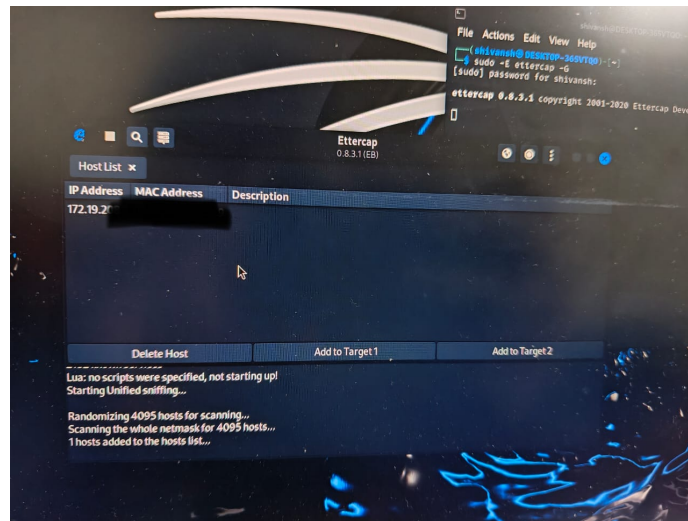


Figure 9: Result

Process 3 : Attack Model and Assumptions

The goal was to perform DNS Spoofing on a mobile hotspot network using tools like Ettercap to intercept DNS queries and redirect traffic. The assumption was that devices would rely on the hotspot's DNS for resolution, allowing ARP poisoning.

Setup and Tools

- Attacker's device: Linux machine running Ettercap or dsniff.
- Victim devices: Multiple devices connected to the mobile hotspot.

Process

1. Connect attacker and victim devices to the hotspot.
2. Launch Ettercap or dsniff for DNS spoofing.
3. Poison DNS cache to redirect traffic to a malicious IP.

Results

The experiment failed due to:

- DNS queries being relayed to a trusted DNS server (e.g., ISP or Google DNS), bypassing the attack.
- Hotspot's DNS relay configuration preventing local poisoning.

Challenges

- DNS relay setup on the hotspot, forwarding queries to external DNS servers.
- Lack of control over hotspot DNS settings.

Conclusion

DNS spoofing was unsuccessful due to the hotspot's DNS relay mechanism. A more configurable network is required for successful DNS poisoning.