Assignment 2 Report

Naman Nirwan

January 25, 2025

1 PART 1: Cracking Hashes with and without Salt

I used around 5,640,000 top passwords to crack the Hashing and retain the passwords following is the result:

1.1 Unsalted Hashing

As instructed in the assignment I used brute force to match calculated hashes with the hashes given. I used some memory overhead to make this faster by using a dictionary to map the hashes for constant lookup in the hash table. Results are following:

1.1.1 MD5

- Total MD5 cracked passwords: 499.
- MD5 cracking completed in 4.25 seconds.

1.1.2 SHA1

- Total SHA1 cracked passwords: 499.
- SHA1 cracking completed in 3.95 seconds.

1.1.3 SHA256

- Total SHA256 cracked passwords: 499.
- SHA256 cracking completed in 4.21 seconds.

1.2 Salted Hashing

As instructed in the assignment I used brute force to add salt text in front of the passwords and then calculate the new hash and match it with an hash in file given to crack. For making the solution faster I used sorted list of passwords according to their commoness, which helped the cracking to be a little faster as I eliminate that salt and hash pair as soon as I crack it. Results are given below:

1.2.1 MD5

- Total MD5 cracked passwords: 499.
- MD5 salted cracking completed in 10.88 seconds.

1.2.2 SHA1

- Total SHA1 cracked passwords: 499.
- SHA1 salted cracking completed in 11.35 seconds.

1.2.3 SHA256

- Total SHA256 cracked passwords: 499.
- SHA256 salted cracking completed in 15.13 seconds.

2 Challenges of Cracking Salted Hashes and How Salts Increase Security

A **salt** is a random value added to a password before hashing to create unique hash values, even for identical passwords. This additional step significantly enhances security by thwarting precomputed attacks, such as rainbow table attacks.

2.1 Challenges in cracking salted Hashes.

- **Increased Uniqueness:** Every password hash is unique because the salt ensures that even identical passwords result in different hash values.
- **Larger Computation Space:** Attackers need to compute hashes for each password-salt combination, greatly increasing the computational effort required.
- **Inefficiency of Parallel Attacks:** Each hash must be cracked independently, making it harder to scale attacks across multiple hashes.

2.2 How Salts Increase Security

- **Prevention of Rainbow Table Attacks:** Salts ensure that precomputed tables for common passwords become ineffective.
- **Individualized Attack Complexity:** Each salted hash requires an attacker to start computations from scratch.
- **Defense Against Database Leaks:** Even if a database is leaked, salts make it harder to deduce plaintext passwords without significant computational effort.

3 MD5 and SHA1 vs SHA256 or SHA3

3.1 MD5 and SHA1 are insecure

- **Collision Vulnerabilities:** MD5 and SHA1 are prone to collision attacks, where two different inputs produce the same hash value. Google demonstrated a practical SHA1 collision attack in 2017. This weakness enables attackers to forge digital signatures and certificates.
- **Speed of Computation:** MD5 was designed for speed, making it vulnerable to brute-force attacks. Also with modern architecture SHA1 brute force attacks are also feasible.
- **Deprecation:** Most organizations and standards (e.g., TLS, certificates) have deprecated MD5 and SHA1 in favor of stronger algorithms.

3.2 Advantages of Using SHA-256 or SHA-3

- **Higher Security Levels:** Both algorithms resist collision, preimage, and second preimage attacks effectively. There is no known collision to SHA256 till today.
- **Increased Hash Length:** SHA-256 produces 256-bit hashes, providing a larger output space compared to MD5 (128-bit) or SHA1 (160-bit). This makes the search space larger for hacking as well.
- **Widespread Adoption:** SHA-256 is part of the SHA-2 family, mandated by many security protocols (e.g., TLS, SSL).

4 Limitations of Relying Solely on Hash Cracking

4.1 Rate-Limiting

- Many systems implement rate-limiting to delay or block repeated hashing attempts. For example, after several failed login attempts, systems may lock accounts or introduce delays.
- Rate-limiting significantly slows brute-force attacks, making them computationally infeasible.

4.2 Cryptographic Protections

- **Keyed Hashing :** HMAC (Hash-based Message Authentication Code) works by combining a secret key with a message and passing them through a cryptographic hash function. The process involves two stages: first, the key and message are hashed together using an inner padding; second, the resulting hash is combined with the key

again using an outer padding. This ensures integrity (the message hasn't been altered) and authenticity (only someone with the secret key can generate or validate the HMAC).

- **Salting:** Adding salts and multiple iterations increases the computational cost of each guess, reducing the attacker's efficiency.
- **Hardware-Based Defenses:** Secure systems may use hardware modules to store secrets securely, preventing extraction of the hash key.

4.3 Hash Iteration Complexity

- Iterative hashing (e.g., PBKDF2, bcrypt) slows down the hash computation process, making brute-force attacks exponentially more difficult. Each iteration builds on the output of the previous hash, creating a chain of computations. This process increases the time required to compute a single hash, as attackers must repeat the entire chain for each password guess. This intentional delay significantly raises the computational cost of brute-forcing hashes.

5 Part 2: Diffie-Hellman Key Exchange Vulnerabilities

5.1 deduce server's private key by solving Discrete Log Problem

Deduced server Private key : 39

In this problem we are given g, y and n and we have to find an x such that $g^x \equiv y \pmod{n}$.

Assumptions Used:

1. If $g = 0$ or 1 then all powers x would only produce 0 and 1 respectively. We can always output 1 in this case. Also observe that without loss of generality, $g < n$. So in the rest of the report we will only consider $g \in \{2 \dots n-1\}$.
2. When n is prime, then $y \neq 0$, as $y = 0$ can not have any solution. So in such cases, $y \in [n-1]$ and $x \in [n-1]$. Notice that one can equivalently take the $x \in \{0, \dots n-2\}$ as $g^0 \equiv g^{n-1} \equiv 1 \pmod{g}$.

I now describe a few algorithms to solve the discrete log problem. Observe that none of these algorithms are truly polynomial time. No efficient algorithm are known for solving Discrete Log Problem in general setting.

5.1.1 Brute Force Algorithm

In this algorithm we simply try all possible x's ¡ n. This takes $O(n)$ time. Here n is a prime.

5.1.2 Baby-Step Giant-Step Algorithm

This algorithm trades off space to improve the running time over brute force. The idea is to write $x = i\lceil n \rceil + j$ for $0 \leq i, j \leq \lceil n \rceil$. For simplicity let $m = \lceil n \rceil$. First compute g^j for all $0 \leq j \leq m$. Then we compute $y \cdot g^{-im}$ for each $0 \leq i \leq m$. Notice that both can be done in $O(\sqrt{n})$ time. If we could find a collision $y \cdot g^{-im} = g^j$, then $x = im + j$. To find this collision efficiently we store all g^j (for all $0 \leq j \leq m$) by hash-sets and when we compute $y \cdot g^{-im}$ we simply check if that value is already present in the set. This finds the collision in $O(\sqrt{n})$ time (as we need to check if the value is present in the set only $O(\sqrt{n})$ times). Overall this algorithm uses $O(\sqrt{n})$ space and time.

5.2 Man in the middle attack on the Diffie-Hellman Protocol

Assume that there is an attacker Eve who wants to evesdrop on Alice and Bob's conversation or worse even to modify their messages. Here's how Eve attacks using the 'Man In The Middle' attack if Alice and Bob use Diffie-Hellman Protocol.

Eve obtains the public g and p, the generator and modulus, that Alice and Bob have decided to use. Now according to protocol Alice chooses a and sends g^a and Bob chooses b and sends g^b. Eve herself chooses an e and generates g^e. She sends g^e to Alice and Bob claiming that she is Bob and Alice respectively. She receives both Alice and Bob's g^a and g^b which they had sent.

Now Alice receives g^e and believes that this message is from Bob. So she creates the key g^{ae}. Similarly, Bob receives g^e and believes that this message is from Alice. So he creates the key g^{be}. Eve computes both g^{ae} and g^{be}. Notice now that Eve has both the keys and has Alice and Bob convinced that they are talking to each other.

Now when Alice sends a message encrypted by the key g^{ae}, Eve decrypts it, reads (and possibly modifies) it, reencrypts the message using g^{be} and sends it to Bob who successfully decrypts it. The message recieved is what Eve sent him and not Alice but Bob believes that the message is from Alice. Eve similarly intercepts messages from Bob to Alice. Thus Alice and Bob think that they are communicating with each other without realizing that there is a 'man' in the middle, Eve, who is evesdropping and possibly modifing their messages.

This attack happens because there is no way for Alice and Bob to authenticate that they are talking to the persons they think they are. This vulenrability is fixed by using digital signatures (a mechanism for authentication) and other such protocols. I have used and tried multiple MITM attack scenarios and processed my results mention below:

5.2.1 Redirect Using iptables

The iptables command is a powerful tool for managing network traffic on Linux systems. It can be used to redirect traffic from one IP address to another, making it useful for Man-in-the-Middle (MiTM) attacks.

- The nat (Network Address Translation) table in iptables can be used to modify the destination of packets.
- Specifically, the **PREROUTING** and **OUTPUT** chains are used to redirect incoming or outgoing traffic:

- **PREROUTING:** Redirects traffic before routing decisions are made (useful when the attacker acts as a router or gateway).
- **OUTPUT:** Redirects traffic generated by the local machine (useful on the client itself).

- Procedure to Redirect the packets:
  - **Redirect Outgoing Traffic (Client-Side Redirection):** This rule modifies outgoing packets on the client machine, redirecting them to attacker destination:

```
$ sudo iptables -t nat -A OUTPUT -p tcp -d <target_ip>
--dport <target_port> -
j DNAT --to-destination <attacker_ip>:<attacker_port>
```

  - **Redirect Incoming Traffic:** This rule modifies incoming packets on a gateway or router to redirect them:

```
$ sudo iptables -t nat -A PREROUTING -p tcp
-d <target_ip> --dport
<target_port> -j DNAT --to-destination
<attacker_ip>:<attacker_port>
```

5.2.2 Modifying the Host File for Traffic Redirection

The /etc/hosts file is a simple text file used by operating systems to map hostnames to IP addresses. By modifying this file, you can redirect traffic intended for a specific hostname or IP address to another IP address, such as an attacker's server.

- When a client tries to access a hostname, the operating system checks the /etc/hosts file before querying the DNS server.
- If a mapping for the hostname is found in the /etc/hosts file, the system uses the specified IP address instead of querying DNS.
- This allows you to redirect requests for specific hostnames to any desired IP address.

5.2.3 ARP spoofing/poisoning:

ARP Spoofing (or ARP Poisoning) is an attack technique where the attacker sends fake ARP (Address Resolution Protocol) messages to associate their MAC address with the IP address of a legitimate device (e.g., a server or router). This allows the attacker to intercept, modify, or redirect traffic in a local network.
How ARP poisoning is performed:

- Enable IP forwarding on the attacker's machine:

```
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

- Use a tool like ettercap to search all the hosts on the network. My hostel network didn't supported this so I figured out it worked in library. After that select Target 1 as Client and Target 2 as Gateway router and inject fake ARP messages by running ARP poisoning MITM attack.

- Intercept and inspect traffic using tools like Wireshark.