SIL765 - Network and System Security

Assignment - 2 Report

Cracking Hashes and Exploring
Diffie-Hellman Key Exchange Vulnerabilities

Anand Sharma (2024JCS2049)
Surajprakash Narwariya (2024JCS2044)
M.Tech Cyber Security, IIT Delhi

PART 1

1. Challenges of Cracking Salted Hashes and Their Security Benefits

The code in attack1.py and attack2.py demonstrates the difference between cracking unsalted and salted hashes. The main challenges and security benefits of salted hashes are:

- Increased Complexity: In attack2.py, each hash is combined with a unique salt. This means that even for identical passwords, the resulting hashes are different due to the salt. This significantly increases the complexity of the cracking process.
- Time and Resource Consumption: Salted hashes require more computational resources to crack. For each password attempt, the cracking algorithm must combine it with the salt before hashing, which increases the time required for each guess.
- Protection Against Rainbow Tables: Salts effectively nullify the use of pre-computed hash tables (rainbow tables), as each salt creates a unique hash even for common passwords.
- Unique Hashes for Identical Passwords: Even if two users have the same password, their hashed passwords will be different due to different salts, enhancing overall security.

2. Security Considerations of MD5, SHA1, and SHA256

The code tests three hashing algorithms: MD5, SHA1, and SHA256. Here's an analysis of their security implications:

a) MD5 (Message Digest algorithm 5):

- Considered cryptographically broken and unsuitable for further use.

- Vulnerable to collision attacks, where two different inputs can produce the same hash.
- The 128-bit hash size is considered too small for modern security standards.

b) SHA1 (Secure Hash Algorithm 1):

- Also considered cryptographically broken.
- Vulnerable to collision attacks, though less severely than MD5.
- The 160-bit hash size offers slightly better security than MD5, but is still inadequate.

c) SHA256:

- Part of the SHA-2 family, currently considered secure.
- Produces a 256-bit hash, significantly more resistant to collision and preimage attacks.
- No known practical attacks that break SHA256.

The progression from MD5 to SHA1 to SHA256 represents an increase in security. SHA256 is currently the most secure of the three and is recommended for use in modern applications.

3. Limitations of Hash Cracking and Additional Security Measures

While the provided scripts demonstrate basic hash cracking techniques, relying solely on these methods has several limitations:

- **Rate Limiting:** The scripts don't account for rate limiting, which is a common security measure that restricts the number of attempts in a given time period.
- **Lack of Cryptographic Protections:** Modern password storage often uses more advanced techniques like bcrypt, scrypt, or Argon2, which are designed to be slow and resource-intensive, making large-scale cracking attempts impractical.
- **Hash Iteration (Key Stretching):** The scripts don't implement hash iteration, where the hash function is applied multiple times to increase the computational cost of cracking.
- **Limited Dictionary:** The effectiveness of the attack is constrained by the dictionary used (rockyou.txt). A more comprehensive or targeted wordlist could yield better results but would also increase computational requirements.
- **No Consideration for Password Complexity:** The scripts don't account for password complexity rules that many systems enforce, potentially wasting time on attempts that don't meet minimum requirements.

PART 2

1. Understanding the Server (Oracle)

2. Client Implementation

The client implementation successfully connects to the oracle server and performs the Diffie-Hellman key exchange. Here's a breakdown of the key components:

1. Server Connection: The client establishes a connection to the oracle server using the provided IP address (10.237.27.193) and port (5555).
2. Entry Number Transmission: The client sends its entry number "2024JCS2049" to the server.
3. Parameter Reception: The client receives and parses the Diffie-Hellman parameters (prime number P and generator G) from the server.
4. Key Generation and Exchange:
    - The client generates a random secret key between 2 and P-1.
    - It computes its public key using the formula: public_key = G^secret_key mod P.
    - The client sends its public key to the server and receives the server's public key.
5. Shared Secret Computation: The client calculates the shared secret using the server's public key and its own secret key.

Vulnerability Exploitation

The client code successfully exploits the vulnerability in the Diffie-Hellman implementation by performing a brute-force attack to deduce the server's private key:

1. Brute-Force Attack:
    - The client iterates through possible secret key values from 2 to P-1.
    - For each value, it computes (client_public_key^i mod P) and compares it to the shared secret.
    - When a match is found, the value of i represents the server's secret key.
2. Results Verification:
    - The code verifies the deduced server secret key by computing the shared secret using this key.
    - It compares the calculated shared secret with the original shared secret to confirm the correctness of the deduced key.

Security Implications

This implementation demonstrates several critical vulnerabilities in the Diffie-Hellman key exchange:

1. Small Prime Number: The use of a small prime number allows for feasible brute-force attacks on the private key.
2. Predictable Parameters: The server generates parameters based on the entry number, which could lead to predictable and potentially weak key exchanges.

Result

- The server private key is 34 for Entry Number 2024JCS2049.
- The server private key is 64 for Entry Number 2024JCS2044.
- The client implementation successfully demonstrates the vulnerability of the Diffie-Hellman key exchange when using small prime numbers. By brute-forcing the server's private key, we highlight the importance of using sufficiently large prime numbers in real-world applications.

3. Exploiting the Vulnerability

Introduction

This report details the implementation and analysis of a Man-in-the-Middle (MITM) attack on the Diffie-Hellman key exchange protocol. The attack exploits vulnerabilities in the protocol's implementation, specifically the lack of authentication, to intercept and potentially modify communication between a client and a server (oracle).

Methodology

We implemented three different approaches for the MITM attack:

1. Proxy Server Approach
2. DNS Server File Approach
3. ARP Spoofing Approach

1. Proxy Server Approach

It details the implementation and analysis of a Man-in-the-Middle (MITM) attack on the Diffie-Hellman key exchange protocol. The attack exploits vulnerabilities in the protocol's implementation, specifically the lack of authentication, to intercept and potentially modify communication between a client and a server (oracle).

The MITM attack is implemented using a proxy server that acts as an intermediary between the client and the oracle server. The proxy server performs two main functions:

1. It communicates with the oracle server, pretending to be a legitimate client.

2. It communicates with the actual client, pretending to be the oracle server.

Implementation Details

The proxy server is implemented in Python and uses the following key components:

1. Socket Programming: The `socket` library is used to establish network connections.
2. Threading: The `threading` library is used to handle concurrent connections with the client and the oracle server.
3. Random Number Generation: The `random` library is used to generate secret keys.
4. Prime Number Verification: The `sympy` library is used to generate and verify prime numbers.

Key functions in the implementation:

- `setupOracleServer()`: Establishes a connection with the oracle server, performs the Diffie-Hellman key exchange, and attempts to brute-force the server's secret key.
- `setupClient()`: Listens for client connections, generates Diffie-Hellman parameters, performs the key exchange with the client, and attempts to brute-force the client's secret key.
- `generatePrime()`: Generates a large prime number for use in the Diffie-Hellman exchange with the client.

Attack Workflow

1. The proxy server initiates concurrent connections with the oracle server and waits for a client connection.
2. For the oracle server connection:
    - It sends the entry number and receives Diffie-Hellman parameters.
    - It generates its own secret key and computes a public key.
    - It performs the key exchange and computes a shared secret.
3. For the client connection:
    - It generates its own Diffie-Hellman parameters.
    - It performs a key exchange with the client.
4. By establishing separate connections and performing independent key exchanges, the proxy can intercept and potentially modify all communications between the client and the oracle server.

Vulnerabilities Exploited

1. Lack of Authentication: The Diffie-Hellman protocol, as implemented here, does not authenticate the parties involved. This allows the MITM to impersonate both the client and the server.
2. Small Prime Numbers: The oracle server uses relatively small prime numbers (based on the entry number), making it feasible to brute-force the secret keys.

Results

The MITM attack successfully:

1. Intercepted the communication between the client and the oracle server.
2. Established separate shared secrets with both the client and the oracle server.
3. Brute-forced the secret keys of both the client and the oracle server.

The ability to determine the secret keys demonstrates a complete breach of the key exchange protocol's security.

2. DNS File Approach

In this method, we simulated a DNS spoofing attack by using a local file to store IP address and port information, which the client reads to connect to the server. The man-in-the-middle (MITM) attacker can modify this file to redirect the client's connection to their own machine.

Implementation Details

1. A file named "dnsLookup.json" was created to store server information:

```
{“serverIp” : “10.237.27.193”, “serverPort”:”5555”}
{“serverIp” : “10.X.X.X”,“serverPort”:”4321”}
```

2. The client script was modified to read from this file:

```
with open("dnsLookup.json", 'r') as file:
        data = json.load(file)

    serverIp = data["serverIp"]
    serverPort = data["serverPort"]
```

3. The MITM attacker can modify this file to redirect the connection:

```
def modifyDNS():
```

```
with open("dnsLookup.json", 'r') as file:
    data = json.load(file)

data["serverIp"] = "127.0.0.1"
data["serverPort"] = 6666

with open("dnsLookup.json", 'w') as file:
    json.dump(data, file)
```

Attack Workflow

1. The attacker runs the script to modify the "dnsLookup.json" file, replacing the legitimate server's IP and port with their own.
2. When the client runs, it reads the modified file and connects to the attacker's machine instead of the real server.
3. The attacker's machine acts as a proxy, forwarding communication between the client and the real server while being able to intercept and modify the Diffie-Hellman key exchange parameters.

Security Implications

This approach demonstrates how DNS spoofing can be used to facilitate MITM attacks on the Diffie-Hellman key exchange:

1. It highlights how attackers can redirect traffic without modifying the client's code.
2. It emphasizes the need for additional authentication mechanisms in the Diffie-Hellman protocol to detect such redirections.

By implementing this simulation, we've demonstrated another vector that attackers could exploit to intercept and manipulate the Diffie-Hellman key exchange.

3. ARP Spoofing Approach

This method involves manipulating the Address Resolution Protocol (ARP) cache of the target systems to redirect network traffic through the attacker's machine.

While we were unable to fully implement ARP spoofing due to network restrictions, we explored the concept and its potential for MITM attacks. Here's what we attempted:

1. We considered using the tool 'arpspoof' to perform ARP cache poisoning, which would allow us to redirect network traffic through our machine.

2. Although we couldn't use these tools in our network environment, we simulated part of the attack in our system.
3. In a full implementation, the attacker would send falsified ARP messages to associate their MAC address with the IP address of the legitimate server and the client.
4. This would cause all traffic between the client and server to pass through the attacker's machine, allowing interception and modification of the Diffie-Hellman key exchange.

1. ARP Cache Poisoning Tools:
   'Arpspoof' is a popular tool used for ARP cache poisoning. It's part of the dsniff package and allows an attacker to intercept traffic on a network by sending out falsified ARP (Address Resolution Protocol) messages. The goal is to associate the attacker's MAC address with the IP address of a legitimate server or client on the local network.
2. Simulating the Attack:
   Since we couldn't use these tools in our actual network environment (likely due to security restrictions or ethical considerations), we simulated parts of the attack. This could involve creating a controlled environment, perhaps using virtual machines, to demonstrate the concept without affecting real network traffic.
3. Falsified ARP Messages:
   In a full implementation, the attack would work as follows:
   - The attacker sends out falsified ARP reply packets to the target client, claiming that the attacker's MAC address is associated with the server's IP address.
   - Similarly, the attacker sends falsified ARP replies to the server, associating the attacker's MAC address with the client's IP address.
   - These messages exploit the fact that most devices accept ARP replies even if they didn't send out an ARP request (known as gratuitous ARP).
4. Traffic Redirection:
   As a result of the poisoned ARP caches:
   - The client sends all traffic intended for the server to the attacker's MAC address instead.
   - The server sends all traffic intended for the client to the attacker's MAC address.
   - The attacker's machine becomes an intermediary for all communication between the client and server.

5. Intercepting the Diffie-Hellman Key Exchange:
   With all traffic passing through the attacker's machine, the attacker can:
   - Intercept the initial Diffie-Hellman parameters (prime number p and generator g) sent from the server to the client.

- Intercept and potentially modify the public keys exchanged between the client and server.
- Establish separate Diffie-Hellman key exchanges with both the client and server, allowing the attacker to decrypt and potentially modify all subsequent encrypted communications.

6. Implications for Diffie-Hellman Security:
   This attack demonstrates a critical vulnerability in the basic Diffie-Hellman key exchange protocol - it doesn't authenticate the participants. While Diffie-Hellman provides confidentiality against passive eavesdroppers, it's vulnerable to active MITM attacks without additional authentication mechanisms.

Understanding this approach highlights the importance of network-level security measures in protecting against MITM attacks on key exchange protocols.