REPORT

ASSIGNMENT:2
SIL 765 - Network and System Security

NAME: VIKASH PANDEY M.TECH: CYBER SECURITY ENTRY NO: 2024JCS2613	NAME: JITESH VIVEK SHAMDASANI M.TECH: CYBER SECURITY ENTRY NO: 2024JCS2043

Part 1: Cracking Hashes with and without Salt

Overview of tasks:

Hash functions are essential in securing data, as they transform input into fixed-length, irreversible outputs, commonly used in password storage and cryptographic applications. However, they become vulnerable when used without proper safeguards like salting or when outdated algorithms like MD5 or SHA1 are employed. These vulnerabilities allow attackers to exploit weaknesses, such as precomputed hash tables, collision attacks, and brute-force techniques. In this task, we explore and exploit these vulnerabilities by attacking unsalted and salted hashes through brute-force and dictionary attacks, demonstrating how insecure or poorly implemented hashing mechanisms can lead to critical security breaches.

Task 1: Cracking Unsalted Hashes

Objective:

The objective of this task was to crack unsalted password hashes generated using three hashing algorithms (MD5, SHA1, and SHA256). The task involved:

1. Loading target hashes from provided files.
2. Using a dictionary of commonly used passwords (containing at least 5 million entries) to generate hashes for each password.
3. Comparing the generated hashes with the target hashes to identify matching passwords.
4. Measuring and analyzing the time taken for each algorithm.

Steps:

1. Loading Target Hashes
    - Hashes for each algorithm were stored in separate files: md5_hashes.txt, sha1_hashes.txt, and sha256_hashes.txt.
    - These files were loaded into memory as sets for efficient lookup.
2. Hash Generation and Comparison
    - A wordlist (rockyou.txt) containing 5 million+ passwords was used as a dictionary.
    - For each password, hashes were computed using the MD5, SHA1, and SHA256 algorithms.
    - Each computed hash was compared with the loaded target hashes to find matches.
3. Cracking Process
    - The hashlib library was used for hashing.

- The time library was used to measure the time taken for each cracking attempt.
- Matched hashes were decoded into plaintext passwords for reporting.

4. Output
   - Cracked passwords for each algorithm were displayed along with the time taken.
   - Uncracked hashes were also listed for further analysis.

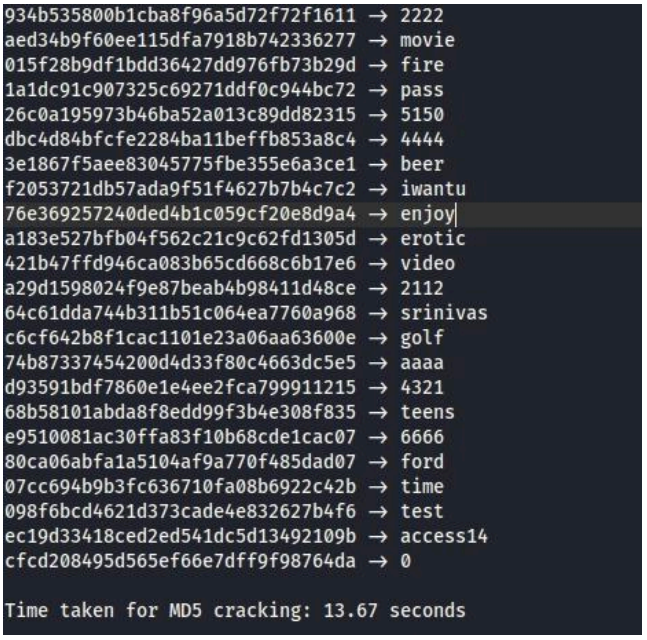
Results:

The table below summarizes the performance and effectiveness of cracking each type of hash.

Algorithm	Total Target Hashes	Cracked Hashes	Time Taken (seconds)
MD5	499	499	13.67
SHA1	499	499	13.47
SHA256	499	499	14.76

Output screenshot:

Unsalted MD5



b)Unsalted sha1

```
94b19a105fce9aa5d2e3270cc428e808538ef548 → movie
1da8402449899ec1ba9c34c095dbb79d0585dcd7 → fire
9d4e1e23bd5b727046a9e3b4b7db57bd8d6ee684 → pass
505e836bb07e69ba387cd3d62a70890b0001bebb → 5150
92f2fd99879b0c2466ab8648afb63c49032379c1 → 4444
600982cf9c0c41e12df616d2a9a72d675345ced7 → beer
0b12fc56d3b2c3f3d153092e951be67e0b2801a5 → iwantu
540e181495e58ae347a7b94e7f43007e0a35a3c1 → enjoy
ab4d8d2a5f480a137067da17100271cd176607a1 → tester
f56d6351aa71cff0debea014d13525e42036187a → 3333
ab874467a7d1ff5fc71a4ade87dc0e098b458aae → 5555
ffbaf58f1231628f9ac2a583f038b51719006ec6 → video
25b732b3425cf8421a84fbe8e6efa1aaf932ad4b → girl
a3cb738850fa39be667c4d6428d72aee854b2cc7 → wolf
a4ac914c09d7c097fe1f4f96b897e625b6922069 → 2000
c692d6a10598e0a801576fdd4ecf3c37e45bfbc4 → bill
612d9ec34bddce122042db4c143e86dca655bc15 → 2112
4036f57732a648e71da3ac2c829c8239a16a4c5d → srinivas
e53d92caa56e00a9cfb84ebfd57dde859f77e2c1 → golf
70c881d4a26984ddce795f6f71817c9cf4480e79 → aaaa
4c1b52409cf6be3896cf163fa17b32e4da293f2e → 6666
d6edd79fd69252cd1adb811d1e99a7398dd6f53a → ford
714eea0f4c980736bde0065fe73f573487f08e3a → time
a94a8fe5ccb19ba61c4c0873d391e987982fbbd3 → test
acfed49ca19dc0bb33b2a8bf56d57aac905922b0 → access14
b6589fc6ab0dc82cf12099d1c2d40ab994e8410c → 0

Time taken for SHA1 cracking: 13.42 seconds
```

c)nsalted sha256

```
61ea0803f8853523b777d414ace3130cd4d3f92de2cd7ff8695c337d79c2eeee → dave
e81dfe69841ad2f7b5790b63e998f0febaf3b29acd732881975130761b98e2c7 → king
d0cfc2e5319b82cdc71a33873e826c93d7ee11363f8ac91c4fa3a2cfcd2286e5 → fred
edee29f882543b956620b26d0ee0e7e950399b1c4222f5de05e06425b4c995e9 → 2222
8a6ba32c9bed6ce703f999f9af6ec23686d44e144e4da572d94c8daca4a9cbab → movie
dc9f28b12dd1818ee42ffc92ecb9403862145988373480d30d3c6c0b7b57e34c9 → fire
d74ff0ee8da3b9806b18c877dbf29bbde50b5bd8e4dad7a3a725000feb82e8f1 → pass
9ff8476644903ba6cb3f90e79bf40b67133a95e44b39123a1d4caf800f216b6b → 5150
79f06f8fde333461739f220090a23cb2a79f6d714bee100d0e4b4af249294619 → 4444
1d8b4cf854cd42f4868849c4ce329da72c406cc11983b4bf45acdae0805f7a72 → beer
eab18a7ebe4a09e361ff2de6c14e0418fe9446403de8abb1da6b0baa9669a522 → iwantu
75fba329cb6ee8907a1f009b69353d8c0e9a8228b25a1095c7976efc1eaa259d → enjoy
9bba5c53a0545e0c80184b946153c9f58387e3bd1d4ee35740f29ac2e718b019 → tester
0cab1c9617404faf2b24e221e189ca5945813e14d3f766345b09ca13bbe28ffc → video
431ba1c6936744f659289a8ff5292f004a51ba6a96dcecc65ef8f0783018dd5f → girl
f76b61b962db075bb76ad6f3fab10f7bd546f92f1b89f18c513d4122575c18ac → wolf
6232101675539 39c87ed8c5f2bfe0b3e0684e12c3a3dd2513613c4e67263b5a1 → bill
44c59909f17c296d6f2ec4a53efac3a951add75aa67616d9c5d9d2f5fbb44f04 → 2112
954c7a1f84096998e6df172544976b0f8357d9d545c0f5ffa0d7ca839ee8a383 → srinivas
625fe74cad4600b5e8b76a9283333eb79052ae50d6af7f660feb4831d87af5d2 → golf
61be55a8e2f6b4e172338bddf184d6dbee29c98853e0a0485ecee7f27b9af0b4 → aaaa
fe2592b42a727e977f055947385b709cc82b16b9a87f88c6abf3900d65d0cdc3 → 4321
5a020aa82f17b6d1d929cead88e4676a97be509ec03774eea6c16da33eacba62 → teens
d7697570462f7562b83e81258de0f1e41832e98072e44c36ec8efec46786e24e → 6666
b2808c947ea04c193e096f4923d13599ab96317dee1e33c8705bfc863f354903 → ford
336074805fc853987abe6f7fe3ad97a6a6f3077a16391fec744f671a015fbd7e → time
8a7be276f8fa620390a6e5e949823fcf0156ba5dc042777e6b6d956bad1c7508 → access14
5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9 → 0

Time taken for SHA256 cracking: 14.76 seconds
```

Observation:

1. Cracking Efficiency
    - MD5: Being a simpler algorithm, it was the fastest to compute and crack. However, it is also less secure due to its simplicity and vulnerabilities.
    - SHA1: Slightly slower than MD5, but still feasible to crack due to its computational similarity.
    - SHA256: The most secure among the three algorithms, requiring significantly more time due to its complexity and longer hash size.
2. Time Complexity
    - The time taken for cracking increased proportionally with the complexity of the algorithm and the size of the wordlist.
    - SHA256 took the longest time, reflecting its computational overhead compared to MD5 and SHA1.
3. Impact of Wordlist Quality
    - The success rate of cracking was highly dependent on the quality of the wordlist (rockyou.txt), as it contains commonly used passwords.

Task 2: Cracking salted Hashes

Objective:

The purpose of this task was to crack salted password hashes generated using MD5, SHA1, and SHA256 algorithms. This involved:

1. Extracting salts and hashes from files containing salted entries in the format salt:hash.
2. Combining each password from a dictionary with the corresponding salt and hashing the result.
3. Comparing the computed hash with the stored salted hash to identify matches.
4. Measuring and analyzing the time taken for cracking each algorithm.
5. Understanding how salts increase the complexity of hash cracking.

Steps:

1. **Loading Salted Hashes**
    - Salted hashes were stored in files (md5_salted_hashes.txt, sha1_salted_hashes.txt, sha256_salted_hashes.txt), each line containing entries in the format salt:hash.
    - The salts and hashes were extracted and stored as a list of tuples for efficient processing.

2. **Loading the Wordlist**
    - The dictionary of passwords (rockyou.txt) was loaded into memory, containing over 5 million entries of commonly used passwords.
3. **Cracking Process**
    - For each salted entry, the password from the dictionary was combined with the salt (as password + salt), and the resulting string was hashed using the corresponding algorithm (MD5, SHA1, or SHA256).
    - If the computed hash matched the stored salted hash, the corresponding password was identified as cracked.

4. **Output**
    - Cracked passwords were displayed along with their salts and hashes.
    - Uncracked salted hashes were also listed.
    - Time taken for each algorithm was recorded and reported.

Result:

The following table summarizes the performance for each algorithm:

Algorithm	Total Salted Hashes	Cracked Hashes	Time Taken (seconds)
MD5	**499**	**499**	**14.72**
SHA1	**499**	**499**	**16.48**
SHA256	**499**	**499**	**19.79**

Output :

a)salted md5

```
Salt: b6ec388d46f08f780fb234be28ad7dcc, Hash: 12e895c5d14012d9d426f941be0897a0 → Password: voodoo
Salt: 61a2e94a9da9ec73a8df3360ab21e7fc, Hash: 2b4ed8dc160c21f2bee96837f31678e5 → Password: magnum
Salt: 9488a8952d63e5cbf37f40d0e2c71739, Hash: 79029db68e61dc7c8766b64472b212bf → Password: juice
Salt: 7b1b5f94a2f8091a2202f8f212d37ba6, Hash: c6a0dc1066ee0bbcf1eb6b0addbb6cdf → Password: abgrtyu
Salt: 45081e325b02fa2a7eba1a7d0780e001, Hash: b87e4828b2016d497cebac4c61b09121 → Password: 777777
Salt: f649a1f5e68321d761b8faec4c9d4a09, Hash: e723229a86d59b3a71edd9fda0c1b7c8 → Password: dreams
Salt: ed0b5e6d928d459b94b50ff8d6381354, Hash: 01074876fe1bf37ca6f53e86d3fea28e → Password: maxwell
Salt: e24230178a85ac48e67ca98495d1c18f, Hash: 9cee83fd1ae283f89b746962eeb08a7d → Password: music
Salt: cf6ac7c2f1fc5133db4fbfd833e8c5fc, Hash: fecf02bd6de85596c848a3001ac3490f → Password: rush2112
Salt: 53e5b672201f7a4d9111989831ebb672, Hash: 9412503d08bc6249b010b22a7382f825 → Password: russia
Salt: f5a150d0b8990d09763fb1c70ebd9461, Hash: cdddd0f06feb15ac9359a302a35f976b → Password: scorpion
Salt: e9cbc8bfb0abfc1432e5efc718691b42, Hash: b96e1b04057c076538d3ecfd81f03a68 → Password: rebecca
Salt: 87c066dc12dffb8ce76e4761b0e14ae8, Hash: f57dd733d25e22f4629fb47169cb6cd5 → Password: tester
Salt: 742c709bce29deccc500897b73e3beaf, Hash: dfecd171a58e6916425ccb032b428be8 → Password: mistress
Salt: 68719c2763b2242f91cb7ab6dbdc2b10, Hash: 65cdf1e636f4cd66b827b765e44e09d4 → Password: phantom
Salt: e1e3de4af22664d8548bf97ff6adee5d, Hash: a9981ad98af63bf831f7782dc4624871 → Password: billy
Salt: f2320137382c2d6768e9d835fe6aa213, Hash: 0e13f0b082858dcda8086d84cecbc660 → Password: 6666
Salt: f367503042551d166ba303962830c947, Hash: 2e63c4a60deea47955457a2b7ce8753e → Password: albert

Time taken for MD5 salted cracking: 14.72 seconds
```

b)salted sha1

```
Salt: 06f5464c9eb3bd02ad83992bf7985cf1, Hash: 741d0562cce362280d577fd21ca54cd419fe6d1c → Password: voodoo
Salt: 4b6323429ea2ef1dc0fbc57217506edd, Hash: ebfc9c09f78213cbcef36f376b3019c7c583f94e → Password: magnum
Salt: 35053db2123c3e58bd882a4bb542ff7f, Hash: 483c75336551afe4b26167a6b5887833f610f930 → Password: juice
Salt: a27dd5615165ab89058f396e6649566e, Hash: 03e83367500883efa4b74a6c47dbff67bd6d1b22 → Password: abgrtyu
Salt: 069ccddd0d7d9936816c206cfe1a4f97, Hash: 9c09df15e805fe1bbb6da00cddb41316ca079c7d → Password: 777777
Salt: 72c671a147146bcff64f05528fba69fa, Hash: ef4624537a72ea458d5e54d2ff174af9f194b9f9 → Password: dreams
Salt: 4898161119c49e340bd100d9428d23ba, Hash: 7b15b9e034aa1d13c6f99786d70a34898ece2914 → Password: maxwell
Salt: 1fb78a39a70900c9cf930662973d8c7b, Hash: 0643f1f7ae30aa53c1aae7d1fa60398d8c253f4d → Password: music
Salt: e1a0d4eac48f013c4512cd59aac016ee, Hash: 68e565f8b268aee8ae10306e494bfd485275f71a → Password: rush2112
Salt: 4d976cc3b38b19e2f7521641e14f590c, Hash: 4b0996e6ce9d5895f545485ad71256efb1d2d3ed → Password: russia
Salt: 9f9fbf226eea1b2158c45b15f0bcbe3c, Hash: 5ceeda8820d026a592b6600d7f89526b5df3fa95 → Password: scorpion
Salt: e8d7f8eb041344abeded128f305d8612, Hash: a189b5b33e4838be0716d8893cd2afae36196e65 → Password: rebecca
Salt: 259fd8a2e04b193d2cc3700694271f26, Hash: d95b4fc07cc1abe861f0b281d94974e4d597c7c1 → Password: tester
Salt: 0f7cac921ecfac8c7beb849d4d679012, Hash: db4ed4a1eefed86c39f00669d504ae41609c1bc2 → Password: mistress
Salt: fe6ed8c341f75b56348c54afbf3bcd19, Hash: 1590288b1dd44214804336900a756b2defd9a9ae → Password: phantom
Salt: 1a1601cf7a6e829a9a165b6994e3d900, Hash: 8bfe6fbc62219f2864c8be416fee02860c8688dd → Password: billy
Salt: c1fdeae2bdf4db32a8943004b971a9b6, Hash: 1316e124de25eeb0caad8969c4a705b915a3c7c9 → Password: 6666
Salt: 2c47cc10c30d5336690b4ddbaf36eaac, Hash: 4c7f9689a9752b2cb301c3e9536c61444a0817ed → Password: albert

Time taken for SHA1 salted cracking: 16.48 seconds
```

c) salted sha256

```
Salt: 346cea4e2104f695558b36489fe447e3, Hash: fa2e4985f72d7994a614dca71468f03f3f1784e9458e6f4ce4bd60477a1d6c4a → Password: magnum
Salt: 70a4245a6fdc9c2463d47d5ffa121cf8, Hash: 4807ec20ab75fd908c6799d2e834c9a80a8fcc4fec069bcb2c4d433f628b71c2 → Password: juice
Salt: 5494a7ffb713cf27194e2938934c27ce, Hash: d24b9b40f1b08222fee2e0738372b585a0e503e8051bcc29862116aabf753837 → Password: abgrtyu
Salt: 48aa44011edeb58852e6216d6f1f09f0, Hash: f94d895c75aa7c7634216c918435b1ddabe5f898cb0a6f8d652d8f75211cd097 → Password: 777777
Salt: d9d7937ef49b6697211176e7f333b289, Hash: 9bd6b5ec544a9ac88243567180f9cb097d57c1ce122a1825e4c1d59af223a6f2 → Password: dreams
Salt: c3bf46d662c5679665fffe85936e13cf, Hash: 262dc7db173f8b8a38a3b3ff645ed54e105e645f4107bf5392bed9a55a8adc4d → Password: maxwell
Salt: 22c3321f0c5919e1ccfcf500a73b6c55, Hash: 464b741e257fb644fec8b1faf98fc72849d761a444887457968b8b6da5377d94 → Password: music
Salt: aa4a9fa64f477273442f78cf3f9ffc75, Hash: e1aaf913e4fbbe510731b71874f5c0b0cea874b64038a9892e15e7eb8ec239fe → Password: rush2112
Salt: 29dec4b70eb869286e608c7d19a25035, Hash: 55c7d9395732a1f18e0a1f9fe5788cba4799f00d4ae369218e0b7c4b17536130 → Password: russia
Salt: df5b450dd36d9a81d0193cded9e35281, Hash: 5aa8c6ee99b2b55912c295e5dd87c63a39b8890b70a78170d06a6e04e02a2d27 → Password: scorpion
Salt: 8ffd0379d6f66c9e9a470d146cf92af6, Hash: 0caf0564bc9535100738cbb49c5f6dcf390015ed9ee00d134301a62d598bb6ae → Password: rebecca
Salt: 1f6a94ee44234b0ebf7912874abe99ef, Hash: 4d665de6096152b0dd2656f65e0db60656d1156e95113ab48aa998c3f7a1ecea → Password: tester
Salt: a1f37f52ab467fc00510f28610153270, Hash: 94101b9815fc43199288e5c8bcd60360420ab5a9ed180800d31ab7bd13ff6933 → Password: mistress
Salt: 5d9a86500a29c9616eee0e5113edbb6b, Hash: 49939d3746e9812e2cc2c2b512ac49d6f77ef72dfaf1aaae5cd0b31573886f34 → Password: phantom
Salt: 02a5b0a7a0241958203b689ba575b4ba, Hash: f34b75418863a7b21d323343f3e87cd6e5e0fde891e02fae0b50ac2b7def732a → Password: billy
Salt: 8caeaa6b8e5b51f18547d8a02ae044fa, Hash: b0a1329448fce045996474f8a1fffaa9dd994446baf3d694f32e6eb4cb1099ba → Password: 6666
Salt: 47b06a27ae14e1f1b03b388ad00621d4, Hash: 52c5859a0e03a7912701876f38cd5a928bd212e737020e2811346a10ca436629 → Password: albert

Uncracked Hashes:

Time taken for SHA256 salted cracking: 19.79 seconds
```

Observation:

1. **Impact of Salting**
    - Salts prevent the use of precomputed hash tables (e.g., rainbow tables), as each password must now be hashed individually with its unique salt.
    - This significantly increases the computational complexity and time required to crack salted hashes.
2. **Algorithm Complexity**
    - **MD5**: While MD5 is computationally the simplest algorithm, the inclusion of salts negates its vulnerability to precomputed attacks. It was the fastest to crack.
    - **SHA1**: More secure than MD5, but still feasible to crack with sufficient computational resources.
    - **SHA256**: The most secure of the three algorithms, taking the longest time due to its computational complexity and the larger hash size.
3. **Parallel Processing**
    - The use of multithreading significantly reduced the overall time required for cracking by leveraging parallel execution of tasks.
4. **Success Rate**
    - The cracking success rate was directly influenced by the quality and comprehensiveness of the wordlist (rockyou.txt).

ANALYSIS OF PART 1:

1. The Challenges of Cracking Salted Hashes and How Salts Increase Security

Challenges of Cracking Salted Hashes

- **Uniqueness of Salts**: Each password has a unique salt, meaning precomputed attacks like rainbow tables are ineffective. Attackers cannot reuse their work on one password for another, even if they are identical.
- **Increased Computational Complexity**: Attackers must compute the hash for every password-salt combination in the dictionary, significantly increasing the time and resources required to crack hashes.
- **Memory Overhead**: Storing salts for each user adds to storage requirements, and the attacker must handle this overhead while attempting to crack hashes.
- **Parallel Cracking Difficulty**: Each hash requires independent computation due to the unique salt, reducing the effectiveness of parallel cracking and requiring linear scaling of resources.

How Salts Increase Security

- **Prevention of Rainbow Table Attacks**: Rainbow tables rely on precomputed hash values for commonly used passwords. Salting renders them useless because the hash for "password123" with salt "abc" differs from "password123" with salt "xyz."

- **Mitigation of Hash Reuse**: Without salts, identical passwords have identical hashes, making it easier to identify and exploit common passwords across different systems. Salts ensure uniqueness.
- **Resistance to Database Breaches**: Even if an attacker gains access to the hash database, cracking salted hashes requires significantly more effort, especially when salts are long and random.

2. Why MD5 and SHA1 are Considered Insecure

Vulnerabilities of MD5

- **Collision Attacks**: MD5 is susceptible to collisions, where two different inputs produce the same hash. This flaw allows attackers to substitute malicious data without detection.
- **Speed**: MD5 is computationally fast, making brute force and dictionary attacks much more feasible.
- **Deprecated Status**: Due to these vulnerabilities, MD5 is no longer recommended for security-critical applications.

Vulnerabilities of SHA1

- **Collision Attacks**: While more secure than MD5, SHA1 is also vulnerable to collision attacks. Google demonstrated a practical SHA1 collision in 2017, marking its deprecation for security use cases.
- **Pre-image Attacks**: Though more computationally expensive than collisions, pre-image attacks are a concern as computing power increases.
- **Computational Feasibility**: SHA1, like MD5, is relatively fast, making it susceptible to brute-force attacks.

Advantages of Using More Secure Algorithms

- **SHA-256**
    - **Stronger Collision Resistance**: No practical collisions have been demonstrated for SHA-256, making it far more reliable.
    - **Longer Hash Length**: SHA-256 generates 256-bit hashes, making brute force attacks computationally prohibitive.
    - **Wide Adoption**: It is a part of the SHA-2 family, widely recommended and supported in modern cryptographic systems.
- **SHA-3**
    - **Different Construction**: SHA-3 uses the Keccak sponge construction, which is inherently more secure against certain attacks than the Merkle-Damgård structure used by MD5, SHA1, and SHA2.

    - **Future-Proof**: Designed to withstand potential vulnerabilities discovered in SHA-2 and other existing algorithms.

3. Limitations of Relying Solely on Hash Cracking

Rate-Limiting

- **Impact**: Many systems implement rate-limiting mechanisms to prevent repeated login attempts, making online brute-force or dictionary attacks infeasible.
- **Bypass Difficulty**: Even with access to a hash database, rate limits on queries to APIs or authentication systems slow down attempts to verify cracked passwords.

Cryptographic Protections

- **Salting and Key Stretching**: Modern systems often use salting combined with key-stretching algorithms like bcrypt, Argon2, or PBKDF2. These algorithms apply multiple iterations of hashing, exponentially increasing the time needed to compute a single hash.
- **Pepper**: Some systems use a pepper (a secret value) stored separately from the hashes, making cracking infeasible without access to both the hash database and the pepper.

Hash Iteration Complexity

- **Purpose**: Iteration increases the time required to compute each hash, making brute-force attacks infeasible even with large computational resources.
- **Example**: bcrypt and Argon2 allow configurable cost factors, which can scale with advancing computational power.

Hardware Limitations

- **GPU and ASIC Cracking**: While attackers often use specialized hardware like GPUs and ASICs to speed up hash cracking, well-implemented rate-limiting, salting, and key-stretching algorithms can neutralize much of this advantage.
- **Energy and Cost**: Cracking becomes cost-prohibitive due to the energy consumption and time required to break well-secured hashes.

Part 2: Exploring Diffie-Hellman Key Exchange Vulnerabilities

Overview of the tasks:

The Diffie-Hellman key exchange is a fundamental cryptographic protocol used to establish a shared secret between two parties over an insecure channel. However, it becomes vulnerable when implemented without authentication, allowing an attacker to perform a Man-in-the-Middle (MITM) attack by intercepting and replacing public keys. Additionally, if the prime number used in the exchange is small, the discrete logarithm problem (DLP), on which its security relies, becomes computationally feasible, enabling brute-force attacks to deduce private keys. In this task, we exploit these vulnerabilities by first performing a brute-force attack to deduce the server's private key due to a small prime and then simulating a MITM attack to intercept and manipulate the key exchange.

Diffie-Hellman Key Exchange Essentials:

- **Prime Number (P)**: A large prime number used as the modulus in the computations.
- **Generator (G)**: A primitive root modulo PPP used as the base for exponential calculations.
- **Private Keys (a and b)**: Secret values chosen by the client and server, respectively.
- **Public Keys (A and B)**: Values computed as $G^a \bmod P$ and $G^b \bmod P$, shared between the client and server.
- **Shared Secret (S)**: The common key computed as:
    - Client computes: $S=B^a \bmod P$
    - Server computes: $S=A^b \bmod P$

The shared secret S is identical for both parties, enabling secure communication.

Steps:

Part 1: Understanding the Server

1. The server generates:
    - A large prime number P.
    - A generator G.
    - A private key b (kept secret).
2. The server shares P, G, and its public key $B=G^b \bmod P$ with the client.

Part 2: Client Implementation

1. **Connecting to the Server**

- A client script was implemented to connect to the server using the provided IP address and port.
- The client sent its entry number to the server, requesting P and G..

2. **Key Exchange**
   - The client selected a random private key aaa.
   - The client computed its public key $A=G^a \bmod P$.
   - The client sent A to the server.
   - The server responded with its public key $B=G^b \bmod P$.
   - Using B, the client computed the shared secret $S=B^a \bmod P$.

Part3: Exploiting the Vulnerability

Method 1:(discussed in detail in next page)

The server's private key b was deduced through brute-force by testing all possible values $1 \leq b \leq max\_attempts$.For each potential b, $G^b \bmod P$ was calculated and compared to the received B.

If a match was found, b was identified as the server's private key

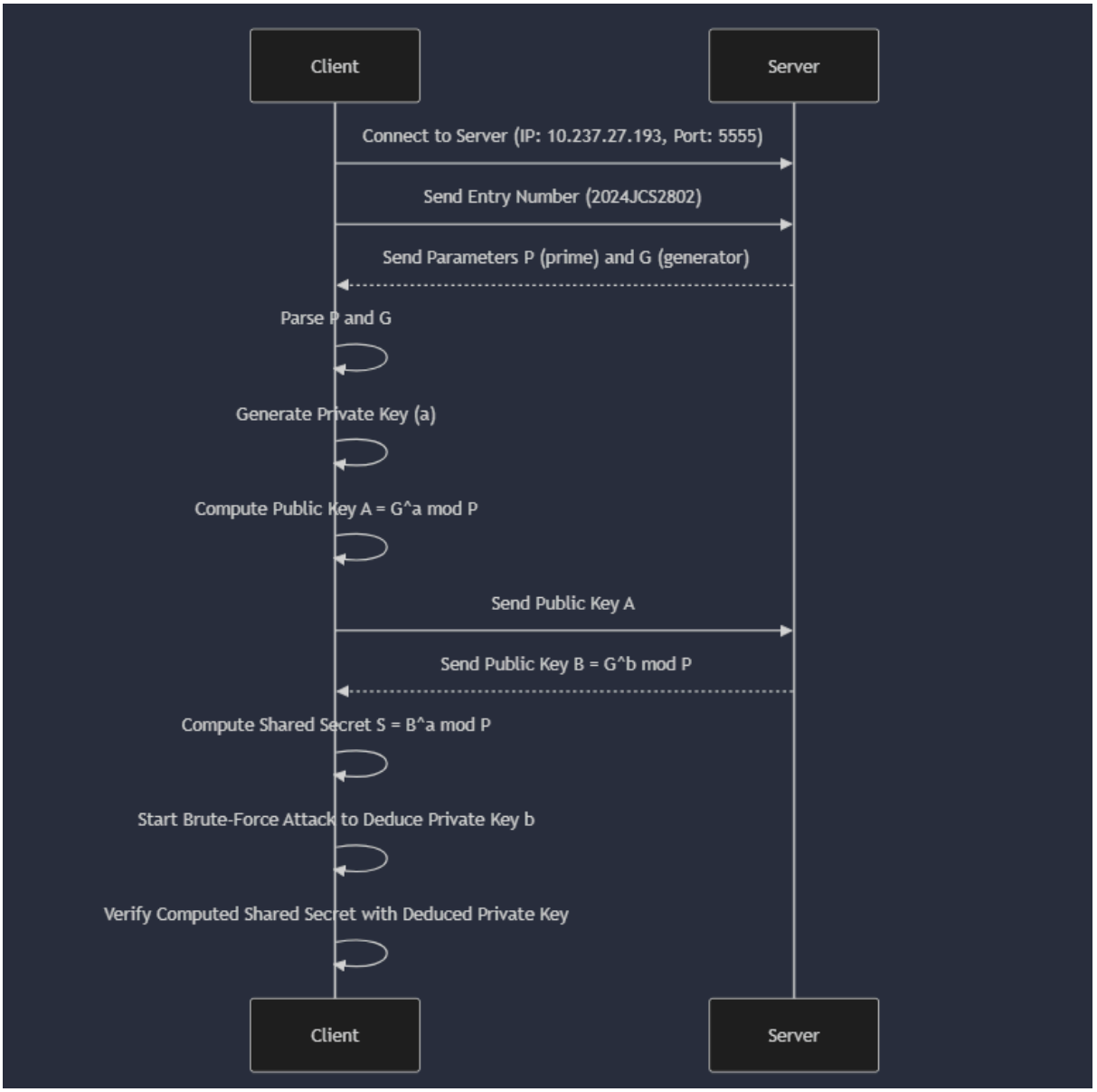
Method 2:(discussed in detail in next page)

simulating a MITM attack to intercept and manipulate the key exchange. We have used 2 approches for it

(A)Approach 1: ARP Spoofing

(B)Approach 2: Client File Modification

METHOD 1:Brute-Force Attack:



Steps:

1. **Connecting to the Server**
    - The client establishes a connection with the server using the provided IP address and port number.
2. **Requesting Parameters**
    - The client sends its unique entry number (2024JCS2802) to the server.
    - The server responds with Diffie-Hellman parameters:
        - P: A prime number.
        - G: A generator.
    -
3. **Parsing Parameters**

- The client parses P and G from the server's response, validating their format.

4. **Generating Private Key**
   - The client generates a random private key (a) within a small range for demonstration.
5. **Computing Public Key**
   - The client computes its public key: $A=G^a \bmod P$.
   - The client sends A to the server.
6. **Receiving Server's Public Key**
   - The server computes and sends its public key: $B=G \bmod P$.
7. **Computing Shared Secret**
   - Using B and its private key a, the client computes the shared secret: $S=B^a \bmod P$
8. **Brute-Force Attack**
   - The client attempts to deduce the server's private key bbb by brute-forcing all possible values:

Check: $G^b \bmod P=B$

   - If successful, b is identified.

```python
def deduce_server_private_key(P, G, B, max_attempts=1000000):

    print("Starting brute force to deduce server's private key...")
    for potential_b in range(1, max_attempts):

        if pow(G, potential_b, P) == B:
            print(f"Server's private key deduced: b = {potential_b} (where B = G^b mod P)")

            return potential_b

        else:
            print(f"failed at b = {potential_b}")

    print("Failed to deduce server's private key within given range.")
    return None
```

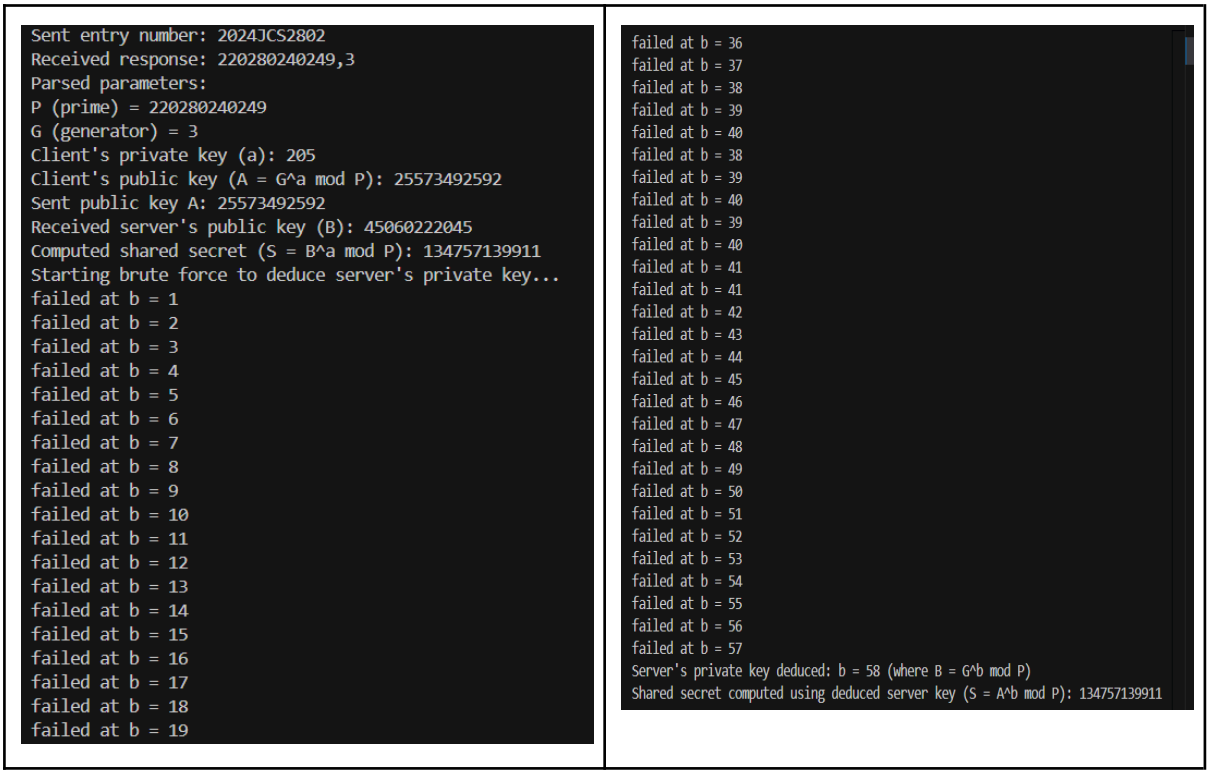
9. **Verifying Shared Secret**
   - The client verifies the deduced private key b by computing the shared secret using: $S=A^b \bmod P$

Challenge:

If prime is sufficiently large (e.g., 2048 bits), brute-forcing becomes computationally very hard

---

Output:



Conclusion for brute force attack:

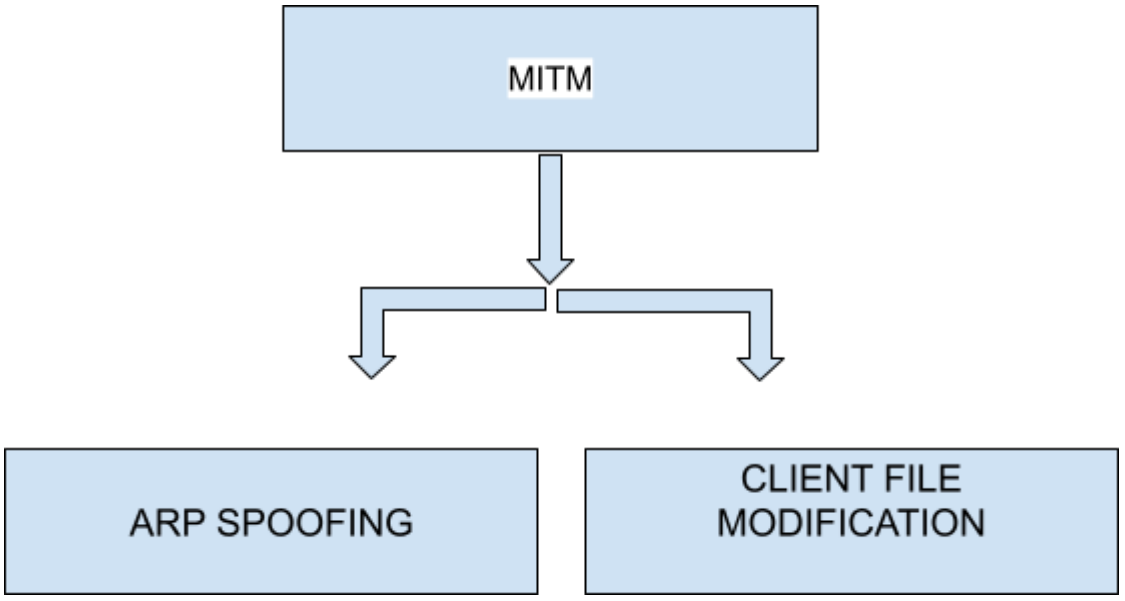
Reason for success of brute force:
The Diffie-Hellman key exchange, while a robust mechanism for establishing a shared secret, is vulnerable when:

1. The prime number P is small.
2. Private keys are selected from a limited range.
3. No additional security measures like authentication are implemented.

This task demonstrated the importance of selecting strong cryptographic parameters and implementing measures to prevent brute-force attacks, such as:

- Using large prime numbers (P) and ensuring private keys (a,b) are chosen from a large range.

Method 2: Man-in-the-Middle (MITM) Attack:



(A)Approach1: ARP SPOOFING

Setup:

- A Virtual Machine (VM) environment was created with three systems:
  - **Client**: Simulated victim machine.
  - **Server**: Simulated target server.
  - **MITM Server**: Configured as the attacker to perform ARP spoofing.
- The systems were connected on the same virtual network using software like VirtualBox.

following command needs to be ran on attacker mitm server before we start arp spoofing

```
sudo iptables -I FORWARD -p tcp --sport 5555 -j NFQUEUE --queue-num 1
```

This command is used to manipulate network packet filtering and routing rules on a Linux system using ip table.

A command-line utility used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. iptables allows administrators to define rules for how incoming and outgoing traffic is handled, including filtering (blocking or allowing traffic), NAT (Network Address Translation), and more.

Implementation:

- **ARP Poisoning**:
  - The MITM server sent spoofed ARP replies to the client and the server:
    - Claimed to be the server when communicating with the client.
    - Claimed to be the client when communicating with the server.
  - This redirected traffic from the client and server through the MITM server.
- **Packet Interception**:
  - Tools like scapy were used to capture and manipulate packets in transit.
  - Traffic was analyzed, decrypted (if unencrypted), and optionally forwarded to maintain seamless communication.
- **Traffic Logging**:
  - Both request and response packets were logged for further analysis of potential vulnerabilities.

Components for this attack

1. client.py

- **Role:** Acts as the client initiating a Diffie-Hellman key exchange with the Oracle server.

- **Functionality:**
    - Connects to the Oracle server.
    - Sends an entry number to request DH parameters (P and G).
    - Generates a private key, computes the corresponding public key (A), and sends it to the server.
    - Receives the server's public key (B) and computes the shared secret (S).
    - Attempts to deduce the server's private key (b) via brute-force.

2. oracle.py

- **Role:** Serves as the Oracle server facilitating the Diffie-Hellman key exchange.
- **Functionality:**
    - Listens for incoming client connections.
    - Receives an entry number from the client.
    - Sends DH parameters (P and G) to the client.
    - Receives the client's public key (A).
    - Computes its own public key (B) using a predetermined private key (b = 49).
    - Sends B back to the client.
    - Computes the shared secret (S = A^b mod P).

3. MITM Server Script

- **Role:** Acts as the Man-in-the-Middle, intercepting and manipulating the communication between the client and Oracle server.
- **Functionality:**
    - Performs ARP spoofing to position itself between the client and server.
    - Intercepts packets on TCP port 5555.
    - Modifies specific data within these packets (e.g., entry numbers and public keys).
    - Forwards the modified packets to their intended destinations, effectively altering the DH key exchange process.

Steps:

1.ARP Spoofing Initiation:

- The MITM server sends ARP replies to both the client and Oracle server, associating each other's IP addresses with the MITM's MAC address.
- **Result:** All traffic between the client and server is routed through the MITM server.

2. Client Sends Entry Number:

- The client (client.py) connects to the Oracle server (oracle.py) and sends the entry number "2024JCS2043".
- **MITM Action:** Intercepts this packet, logs the payload, modifies the entry number to "2024JCS2613", and forwards the altered packet to the Oracle server.

3. Oracle Server Responds with DH Parameters:

- The Oracle server sends "432020424503,5" as DH parameters (P and G).
- **MITM Action:** Intercepts this response, possibly logs it, and forwards it unaltered (assuming no modification logic for DH parameters).

4. Client Generates and Sends Public Key (A):

- The client generates a private key (a = 658), computes A = G^a mod P = 335700373639, and sends A to the server.
- **MITM Action:** Intercepts this packet, logs the payload, modifies the public key to an altered value (e.g., "33570037363"), and forwards the modified packet to the Oracle server.

5. Oracle Server Computes and Sends Public Key (B):

- The Oracle server computes B = G^b mod P = 135123667066 using its private key (b = 49) and sends B to the client.
- **MITM Action:** Intercepts this packet, logs the payload, modifies the public key to an altered value (e.g., "13512366706"), and forwards the modified packet to the client.

Working in detail:

1. Network Setup and ARP Spoofing

Before any communication occurs, the MITM server sets up its position in the network:

- **ARP Spoofing:**
    - The MITM server continuously sends fake ARP replies to both the client and the Oracle server.
    - **To the Client (10.0.2.102):**
        - Associates the Oracle server's IP (10.0.2.100) with the MITM's MAC address (08:00:27:68:e2:97).
    - **To the Oracle Server (10.0.2.100):**
        - Associates the client's IP (10.0.2.102) with the MITM's MAC address (08:00:27:68:e2:97).

- **Effect:** Both the client and server believe that the MITM server is the other party, routing all their traffic through the MITM.

2. Initiation of the Diffie-Hellman Key Exchange

With ARP spoofing in place, the client begins the DH key exchange:

- **Client (client.py):**
  1. **Connects to Oracle Server:**
     - Establishes a TCP connection to 10.0.2.100 on port 5555.
  2. **Sends Entry Number:**
     - Transmits the string "2024JCS2043" to request DH parameters.
  3. **Receives DH Parameters (P and G):**
     - Receives a response like "432020424503,5".
     - Parses P = 432020424503 (prime number) and G = 5 (generator).
  4. **Generates Private and Public Keys:**
     - **Private Key (a):** Randomly selects a number (e.g., 658).
     - **Public Key (A):** Computes A = G^a mod P = 5^658 mod 432020424503.
  5. **Sends Public Key (A) to Oracle Server:**
     - Transmits the computed public key A to the server.
  6. **Receives Server's Public Key (B):**
     - Receives B = 135123667066.
  7. **Computes Shared Secret (S):**
     - Calculates S = B^a mod P = 135123667066^658 mod 432020424503.

3. Oracle Server's Response

The Oracle server processes the client's request:

- **Oracle Server (oracle.py):**
  1. **Listens for Client Connections:**
     - Awaits incoming TCP connections on 10.0.2.100:5555.
  2. **Receives Entry Number:**
     - Accepts the entry number "2024JCS2043" from the client.
  3. **Sends DH Parameters (P and G):**
     - Transmits "432020424503,5" to the client.
  4. **Receives Client's Public Key (A):**
     - Receives A = 335700373639.
  5. **Computes and Sends Server's Public Key (B):**
     - **Private Key (b):** Predetermined as 49.
     - **Public Key (B):** Computes B = G^b mod P = 5^49 mod 432020424503 = 135123667066.

        - Sends B to the client.
    6. **Computes Shared Secret (S):**
        - Calculates S = A^b mod P = 335700373639^49 mod 432020424503 = 208267566443.

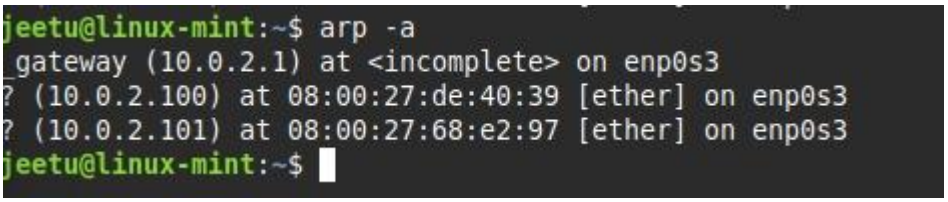
4. MITM Server Interception and Modification

As the DH key exchange occurs, the MITM server intercepts and modifies the communication:

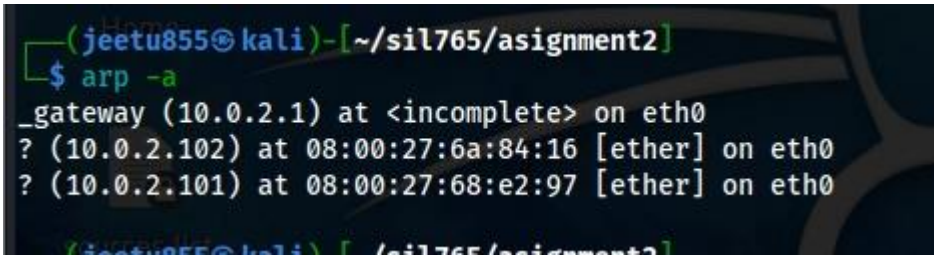
- **Packet Interception:**
    - The MITM server, positioned between the client and Oracle server due to ARP spoofing, captures all packets exchanged between them on port 5555.
- **Packet Modification:**
    - **Client to Server:**
        - **Original Payload:** "2024JCS2043"
        - **Modified Payload:** "2024JCS2613"
        - **Original Public Key (A):** e.g., "335700373639"
        - **Modified Public Key (A):** "33570037363" (altered value)
    - **Server to Client:**
        - **Original Public Key (B):** "135123667066"
        - **Modified Public Key (B):** "13512366706"
    - **Process:**
        - The MITM script identifies packets based on their direction (client to server or server to client) and destination/source ports.
        - When specific strings (e.g., entry number or public keys) are detected in the payload, it replaces them with altered values.
        - Sends the modified packets to the intended recipient while dropping the original, unmodified packets.
- **Effect of Modification:**
    - **Entry Number Change:**
        - Alters the client's identification or request, potentially causing the server to process a different entry.
    - **Public Key Alteration:**
        - By modifying the public keys, the MITM can influence the shared secret computation.
        - This could lead to the MITM being able to derive the shared secret (S), enabling it to decrypt or manipulate further communication.

Outputs:

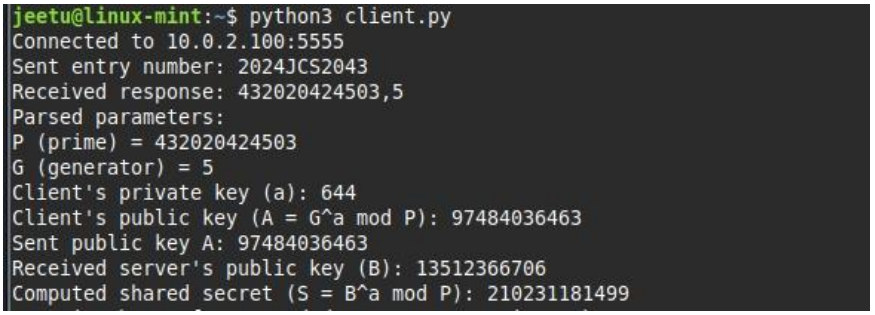
initial client arp -a



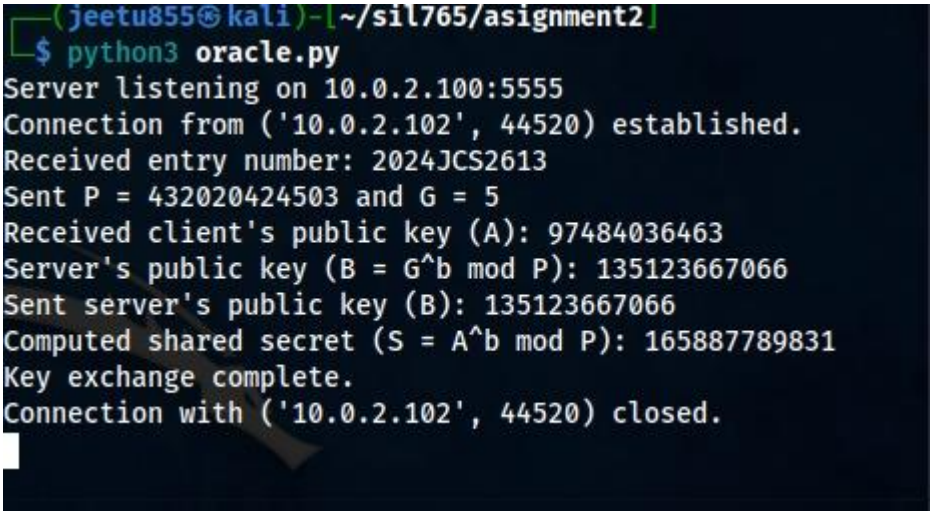
Initial oracle arp -a



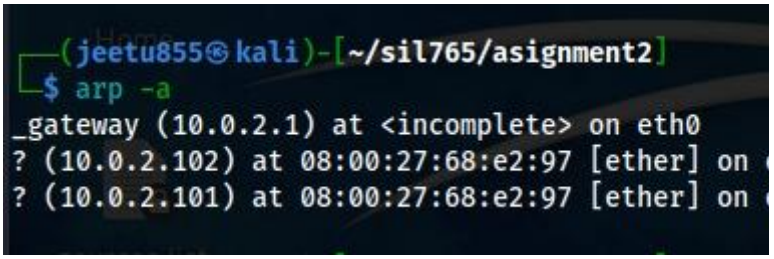
client script run



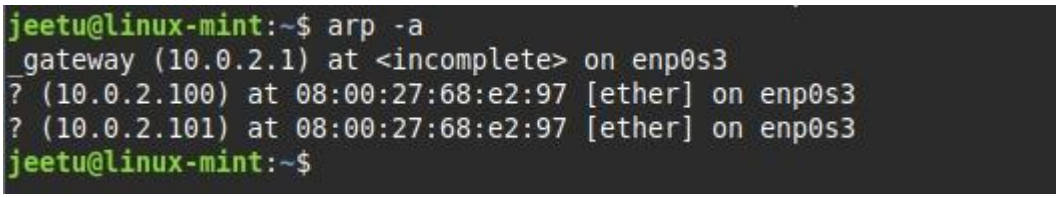
oracle script run



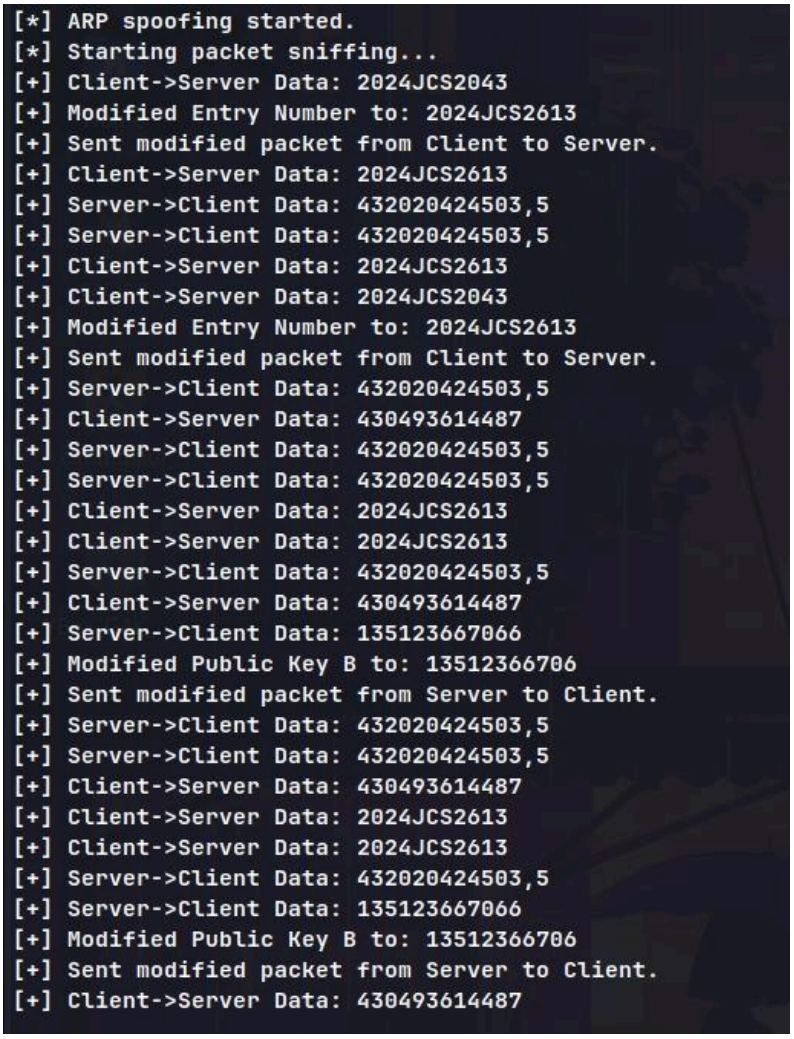
oracle arp -a after spoofing



server arp -a after spoofing

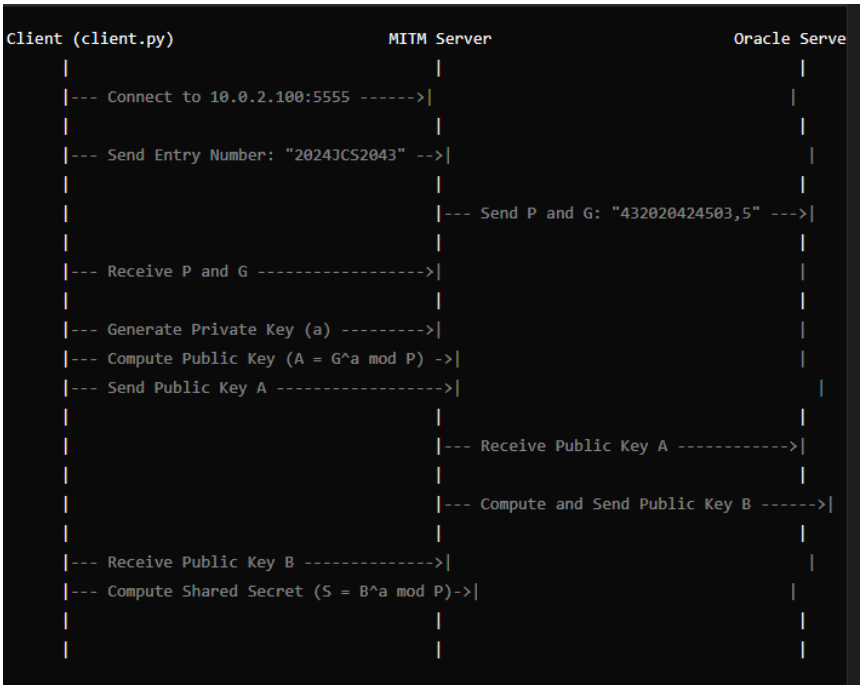


mitm script output

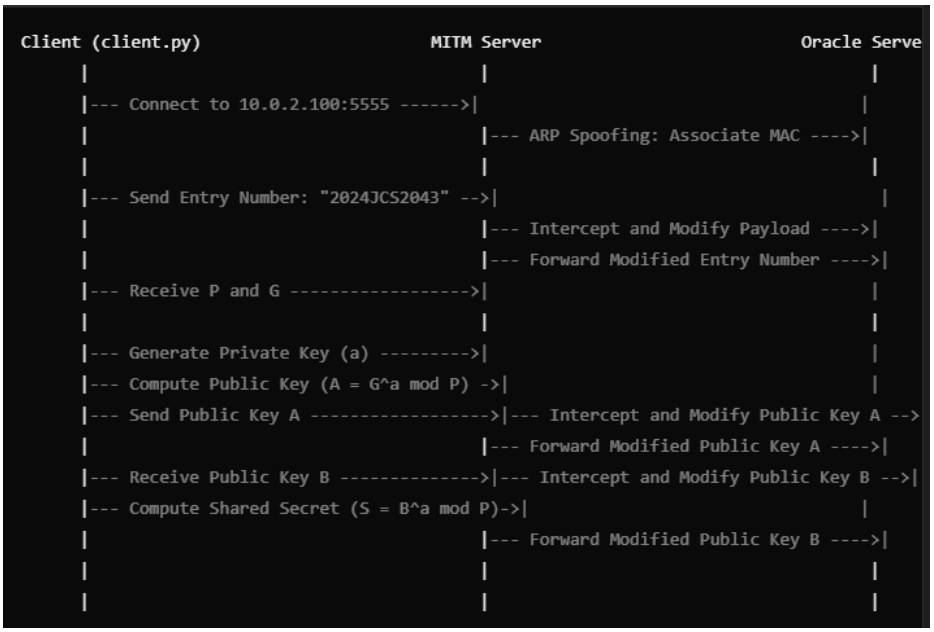


Flow of work:

a)Simple working (if not modified)



b)Working with MITM interception modification



Outcome:

- The MITM server successfully intercepted all traffic between the client and the server.
- Sensitive information such as session keys, plaintext data, or credentials was extracted from unencrypted communication.

Challenges:

During the process of executing the MITM attack, several challenges were encountered, particularly in achieving seamless packet interception and modification. Ensuring the integrity of modified packets was a significant hurdle, as improper checksum recalculations or incomplete packet reconstruction led to discarded or unprocessed packets. Additionally, the simultaneous forwarding of both original and modified packets caused inconsistencies in communication, requiring careful adjustments to drop unmodified packets after modification. Timing issues in ARP spoofing also posed difficulties, as maintaining consistent ARP poisoning across devices demanded precise intervals and synchronization. Finally, understanding and manipulating the intricate Diffie-Hellman exchange process highlighted the complexity of selectively altering data without disrupting the overall flow, necessitating meticulous debugging and packet-level analysis using tools like Wireshark and Scapy.

Conclusion for arp spoofing:

the successful execution of the MITM attack demonstrates a critical vulnerability in the Diffie-Hellman (DH) key exchange protocol: its lack of authentication mechanisms, which allows an attacker to intercept and manipulate messages between parties without detection. This weakness arises because DH exchanges focus solely on securely establishing a shared secret but do not verify the authenticity of the communicating entities or the integrity of the exchanged data. To mitigate this vulnerability, implementations should incorporate robust authentication methods, such as digital signatures or certificates (e.g., integrating DH within TLS protocols like HTTPS), which verify the identities of the parties involved and ensure message integrity, effectively preventing unauthorized interference.

(B) Approach2: CLIENT FILE MODIFICATION

Setup:

- The given Oracle server by TA,and the client were simulated on the my machine.

- The client file was modified to act as a relay or MITM agent between the client and the server.

Implementation:

- The client script was altered to intercept the outgoing requests from the client to the server.
- Requests and responses were logged, manipulated, and forwarded as needed.
- Key exchange parameters and communication data were intercepted, enabling the analysis of vulnerabilities such as insecure key exchange or plaintext data transfer.

Steps:

Step 1: Intercept Client Requests

- The modified client code redirected requests intended for the trusted server to the malicious server:

```
SERVER_IP = "192.168.1.10"  # Trusted server
SERVER_PORT = 8080
```

Modified Client Code:

```
SERVER_IP = "192.168.1.50"  # Malicious server
SERVER_PORT = 9090
```

- This ensured all outgoing requests from the client were sent to the attacker-controlled server.

Step 2: Simulate Malicious Server Behavior

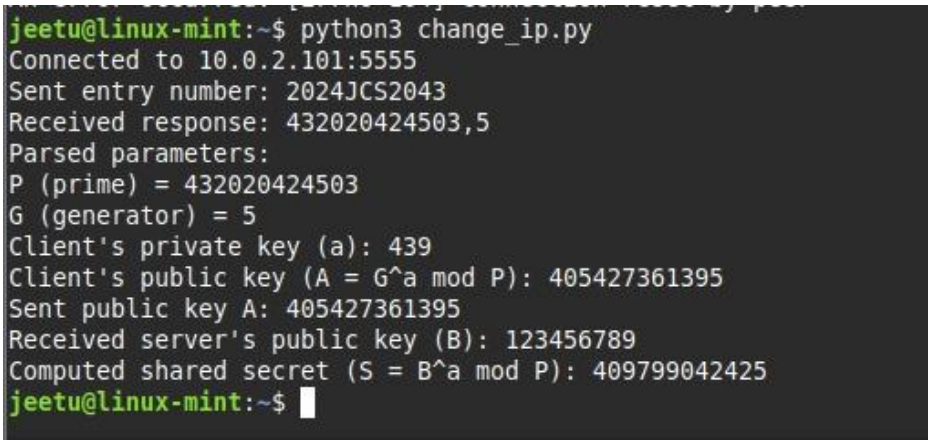
- The malicious server acted as an intermediary:
  1. Received the request from the client.
  2. Logged and analyzed the request data.
  3. Forwarded the request to the original server (optional).
  4. Received the response from the original server.
  5. Modified the response data and sent it back to the client.

Step 3: Data Manipulation

- The attacker:
    - Log sensitive information, such as credentials or session keys.
    - Modify requests or responses (e.g., inject malicious payloads, alter transaction details).
    - Disrupt communication by dropping or altering packets.

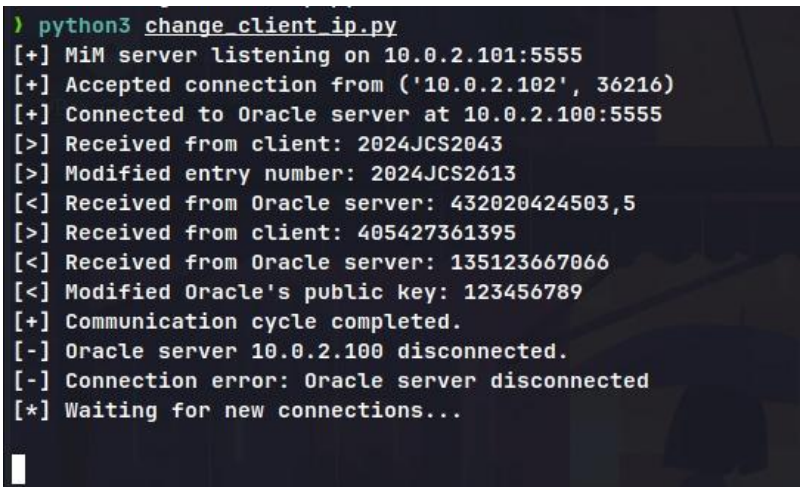
Output:

client output



oracle output

mitm server output



Challenge:

A **Client File Modification Attack** involves directly altering the client code or configuration files to redirect traffic or introduce new behavior, such as pointing the client to a Man-in-the-Middle (MiM) server. The challenges include maintaining compatibility with the original client functionality to prevent detection,

---

Reason Why It Is Possible:

This attack is feasible due to:

1. **No Authentication or Validation**:
    - The client does not validate the server's identity, allowing it to trust the malicious server.
    - Lack of certificates or secure handshakes (e.g., TLS/SSL) makes redirection undetectable.
2. **Hardcoded Server Details**:
    - The client relies on hardcoded IP and port values for communication, which are easily modified.
3. **Plaintext Communication**:
    - If the client-server communication uses plaintext protocols (e.g., HTTP), data is easily intercepted and altered.
4. **No End-to-End Encryption**:
    - Without encryption, the attacker can view and modify sensitive information without detection.

Conclusion for client code modification attack:

This attack demonstrated the risks of insecure communication protocols and improper client-server validation:

1. **Lack of Authentication**: The client trusted the malicious server because it did not validate the server's identity using mechanisms like TLS/SSL certificates.
2. **No Encryption**: Plaintext communication made it easy to intercept and manipulate data.

Use certificates and cryptographic handshakes to ensure the client connects to the intended server to prevent this.