Intro to Cybersecurity - Lab 2 Group 2

Aymane Sghier, Ulukbek Mambetov, Zhipeng Yang

September 2025

1 Introduction

2 Section 4.1 - MD5 Hash Collisions

I have attached a picture of the selfextract archive and files that I was able to create using the pack3 command. I was unable to upload package1.exe and file2 since they were both over 1,900,000 KB.

3 Section 4.2 - SHA-1 Hash Collisions

I made basic good.jpg and bad.jpg image files that I then uploaded to the online SHA1 collider tool and generated good_SHA.pdf and bad_SHA.pdf which both had the same SHA Checksums but different MD5. I then converted the .jpg images to .pdf without SHA and ran the command 'python3 collide.py good.pdf bad.pdf' to generate files out-good.pdf and out-bad.pdf. These two new files also had the same SHA checksums.

4 Section 4.3 - SSL/TLS Certificates

I attempted to generate 384 bits RSA private key by OpenSSL, but it is not allowed because it is extremely insecure. Therefore, I use the Python package rsa to generate 384 bits private key. Then, I use the private key to generate the public key and create the certificate by openssl. After that, I got the modulus and converted it to decimal by python.I use msieve153.exe to find prime 1 and prime 2, and they match the 384 bits host key.

5 Section 4.4 - Kicking it up a notch: 1024-bit RSA keys

5.1 4.4.1 - Attacking the RSA modulus from a different angle

I attempted to factor the two 1024-bit RSA moduli using a GCD attack based on the hint "mind your p's and q's" which suggests looking for shared prime factors.

5.1.1 Attack Methodology

The GCD attack exploits weak RSA key generation where multiple keys accidentally share prime factors. If two RSA keys share a prime factor p, then gcd(n1, n2) = p, allowing instant factorization.

5.1.2 Implementation and Results

I implemented the GCD attack in Python:

```
import math

n1_hex = "
    a4482ebb0580823df76ce4ebbc259c45bfdb51ac75eca9e8e372fc7c7cbdf5ea2415d5e7de75
    "

n2_hex = "
    a7379c4c73718c01c5215a1fb7b61a183d0f4480d2d1f3a4d77f310ded7ccd27c93bacf52143
    "

n1 = int(n1_hex, 16)
n2 = int(n2_hex, 16)

gcd_result = math.gcd(n1, n2)
print(f"GCD = {gcd_result}")
```

Result: GCD = 1, indicating no shared prime factors between the keys.

5.1.3 Conclusion

The GCD attack failed, suggesting these RSA keys were properly generated with distinct prime factors. This demonstrates that not all RSA implementations are vulnerable to this attack method.

5.2 4.4.2 - Decrypting SSL/TLS traffic

I analyzed the tlsdump.pcap file using Wireshark to identify which RSA key was used and attempt traffic decryption.

5.2.1 Traffic Analysis

Using Wireshark with filter tcp.port == 44330, I identified 20 packets of SSL/TLS communication containing a complete handshake and encrypted application data.

5.2.2 Certificate Identification

Analysis of the server certificate revealed the RSA modulus starting with a7379c4c73718c01c5215a1f..., which matches the second RSA public key from section 4.4.1.

5.2.3 Decryption Status

Due to the failed factoring in section 4.4.1, no private key could be reconstructed. The 1788-byte encrypted conversation remains undecrypted.

6 Word Problems

6.1 1. Attack Techniques Summary

The primary technique attempted was the GCD attack on RSA moduli, which exploits shared prime factors between different RSA keys due to weak random number generation during key creation.

6.2 2. TLS Attack Scope

A successful RSA factorization would compromise all TLS sessions using that key pair - past, present, and future. The attack affects the entire cryptographic identity of the server until key rotation occurs.

6.3 3. Defense Strategies

To defend against RSA factorization attacks: use proper random number generation, implement minimum 2048-bit keys, regular key rotation, forward secrecy with ephemeral key exchanges, and migration to post-quantum cryptography.

6.4 4. Factoring Capability Estimate

With standard consumer hardware, 1024-bit RSA keys remain computationally infeasible without specialized resources or cloud computing platforms.

6.5 5. Non-RSA Key Exchange Requirements

For DH/ECDH key exchanges (TLS 1.3+), attackers would need session-specific ephemeral private keys, server signing keys, implementation vulnerabilities, or quantum computing capability.