Semester Learning Portfolio

Alexander Schandorf Sumczynski

November 03, 2025

Abstract

this is the firewall-friendly

Contents

1 Authentication + Link Layer Security: Lecture One

1.1 Notes leacure one

Notes. (Remote User - Authenticationusing Asymmetric Encryption):

$$\begin{aligned}
&\text{A} \rightarrow \text{AS} : ID_A \parallel ID_B \\
&\text{AS} \rightarrow \text{A} : \text{E}(PR_{as}, [ID_A \parallel PU_a \parallel T]) \parallel \text{E}(PR_{as}, [ID_B \parallel PU_b \parallel T]) \\
&\quad \text{A} \rightarrow \text{B} : \text{E}(PR_{as}, [ID_A \parallel PU_a \parallel T]) \parallel \text{E}(PR_{as}, [ID_B \parallel PU_b \parallel T]) \parallel \text{E}(PU_b, \text{E}(PR_a, [K_s \parallel T]))
\end{aligned}$$

$$\begin{aligned}
&\text{A} \rightarrow \text{KDC} : ID_A \parallel ID_B \\
&\text{KDC} \rightarrow \text{A} : \text{E}(PR_{\text{auth}}, [ID_B \parallel PU_b]) \\
&\quad \text{A} \rightarrow \text{B} : \text{E}(PU_b, [N_a \parallel ID_A]) \\
&\text{B} \rightarrow \text{KDC} : ID_A \parallel ID_B \parallel \text{E}(PU_{\text{auth}}, N_a) \\
&\text{KDC} \rightarrow \text{B} : \text{E}(PR_{\text{auth}}, [ID_A \parallel PU_a]) \parallel \text{E}(PU_b, \text{E}(PR_{\text{auth}}, [N_a \parallel K_s \parallel ID_A \parallel ID_B])) \\
&\quad \text{B} \rightarrow \text{A} : \text{E}(PU_a, [N_b \parallel \text{E}(PR_{\text{auth}}, [N_a \parallel K_s \parallel ID_A \parallel ID_B])]) \\
&\quad \text{A} \rightarrow \text{B} : \text{E}(K_s, N_b)
\end{aligned}$$

1.2 Network Security Assignment part 1

Assignment.

Objective: Research and write a concise paragraph about techniques used to mitigate ARP spoofing and Spanning Tree Protocol (STP) attacks (Layer 2 attacks). Please write details on how the chosen technique detects and prevents the attack, and any potential limitations they may have in a network environment. Please also mention your opinion about the complexity of the techniques you found.

Answer. (ARP Spoofing Mitigation):

1) **Static ARP entries:** Using static entries in the ARP table means the IP–MAC mapping cannot be altered by ARP spoofing. The limitation is that when a new device joins the network, its IP–MAC pair must be manually added to the ARP tables of the relevant devices. its nice that i can setup this in a static mac adreses but let say that i ahve to do this for a capnut and maitnign this so alle vesties are update date and
2) **Dynamic ARP Inspection (DAI):**Is a technique where the switches are configured to map each device in the network to a specific IP–MAC pair. If an ARP spoofing attack occurs, then the switch detects that there is an unauthorized ARP request. The limitation of this method is that the switch must be set up with DAI and must be a supported type of switch. This is a better solution than the static assigning since there is a dynamic system in the switches that can help manage the ARP spoofing attacks instead of manually setting each device.
3) **XArp:** Is an anti-spoofing software that can detect if an ARP spoofing attack is being performed on a target system that has installed the XArp on the system, and this is the limitation—that I have to install the XArp and make sure that it's up to date and has no vulnerabilities in this program.

Answer. (STP Attacks Mitigation):

1) **BPDU Guard** is a security feature that automatically puts a PortFast-configured access port into an error-disabled state when it receives any BPDU, protecting the STP domain from rogue switches or misconfiguration
2) **Root Guard** is a security feature that prevents non-root ports from becoming root ports by placing them into a root-inconsistent state if they receive superior BPDUs, ensuring the STP topology remains stable and protecting the network from rogue root bridge elections.

1.3 Network Security Assignment part 2

Assignment.

Objective: In this assignment we are going to emulate a Man-in-the-Middle (MITM) attack using this network topology.

As an attacker we should connect to the switch to be able to communicate with the target/ victim hosts. From now on, we refer to our two targets hosts as victims.

Answer. (Experiencing Layer 2 attacks):

1) The setup I have is two lightweight Lubuntu systems and a Kali Linux where the Man-in-the-Middle attack will be performed. The network is connected to a NAT network through my local machine.

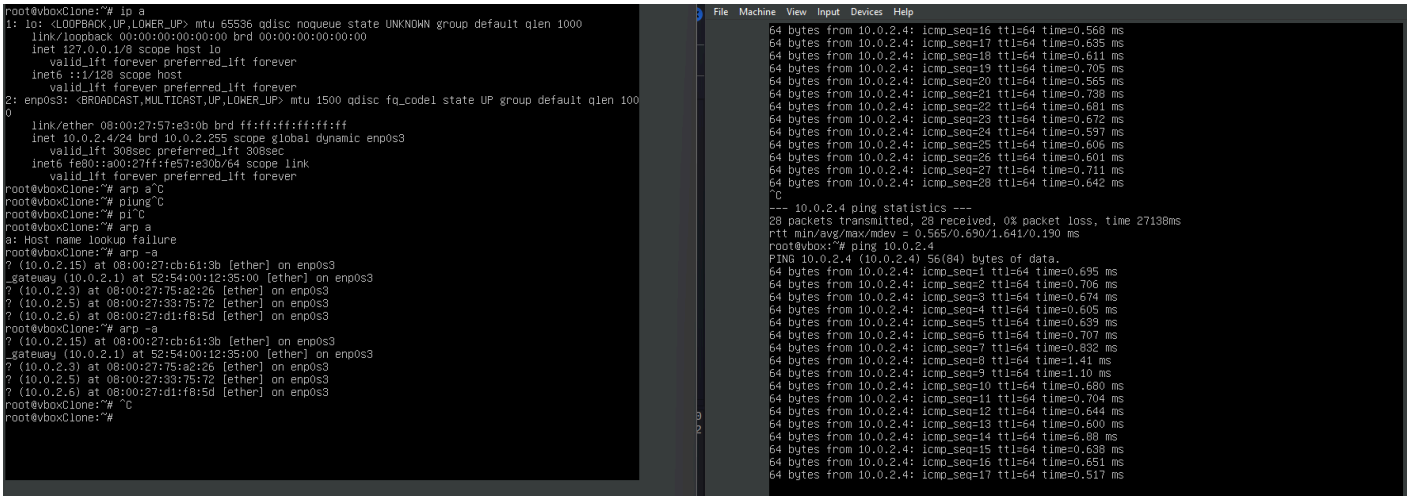

Figure 1: The two Lubuntu machines

In Figure 1 there are the two lightweight Lubuntu machines. The right machine is performing a ping to the other machine (on the left), and the left machine is running the arp -a command to show the devices that are running on this NAT network.

2) The next step is to perform the ARP spoofing attack on the two targets. To do that, on the Kali machine I use the program Ettercap to scan for the two targets and select them as victims, where it will then perform the spoofing attack.

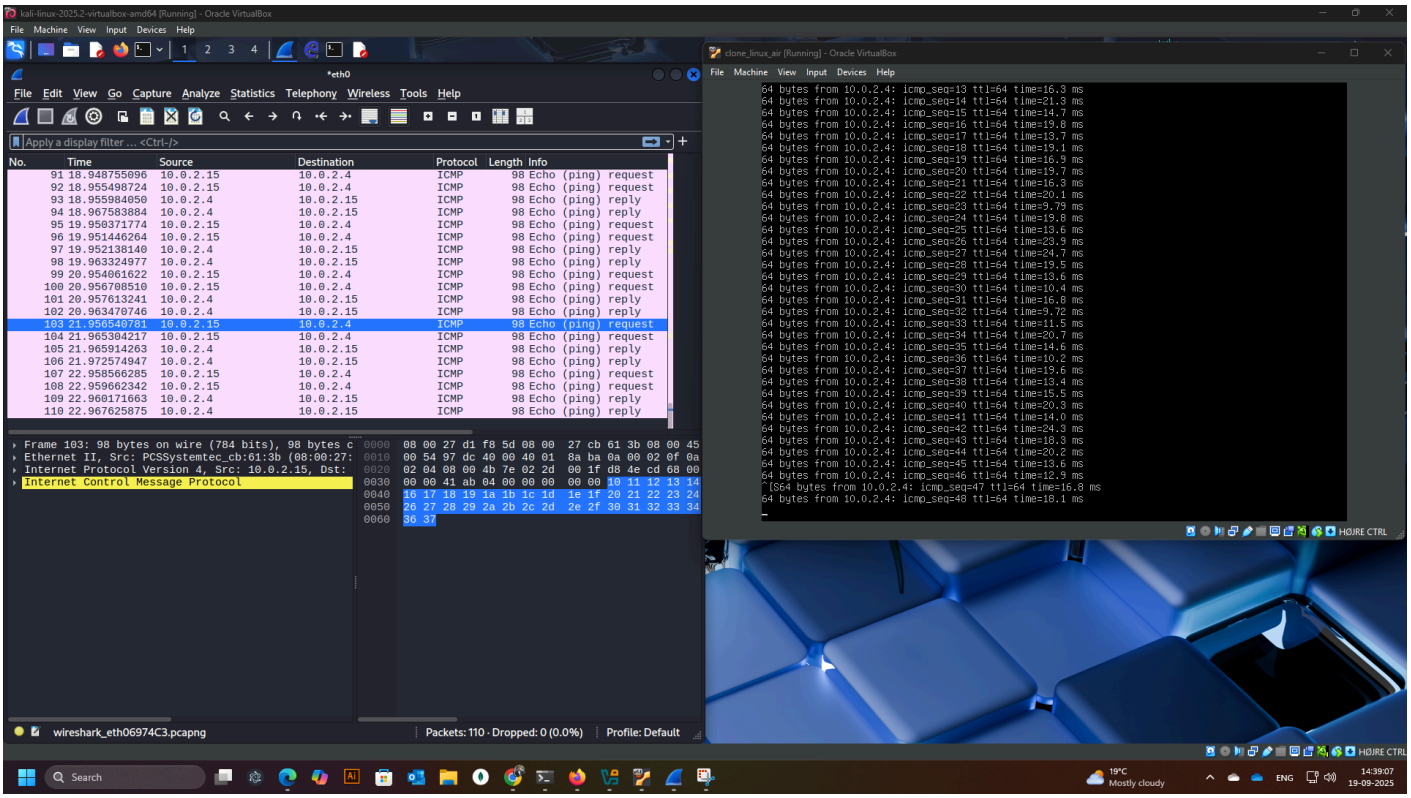

Figure 2: The two Lubuntu machines

In Figure 2, it shows how the attack is under execution, where on the left is the Lubuntu machine that performs a ping to the other Lubuntu machine (on the right). But since we have created a Man-in-the-Middle between the two targets, the traffic can now be seen on the Kali machine, as shown in the image. In this, Wireshark is capturing the traffic between the two machines.

2 TCP/IP Internet Layer Security

2.1 Assignment Experiencing IPsec (Group) part one

Group:

Alexander Sumczynski, Marcus Kolbe, Luca,

Task 1: In the firt part of the start is setting up the two system ubunto severs that suold communicate toghter,

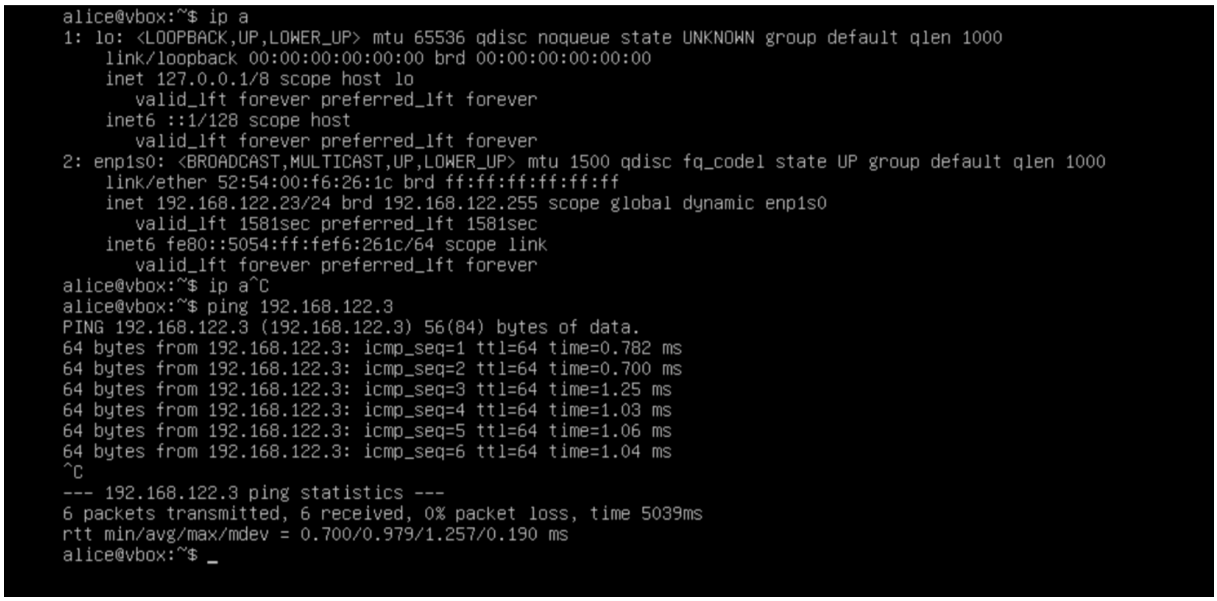

Figure 3: The two Lubuntu machines: alice and bob

In Figure 3 shows how after sinnign up the config files that alcie macinge can ping the bob virtuel maicnge

Task 2 Pre-IPsec Capture:

In Task 2, setting up the capture traffic between the two virtual machines will first happen after some traffic has been passed through the system. Observing these packets being sent is just normal traffic that is not encrypted or anything. I can see the GET request to the Bob machine that is hosting an Apache2 service, so all the TCP handshakes and the GET/response is plain text

Task 3 Capturing IKE:

Now starting tshark, then launching the IPsec services. This will allow the capture of the IKE (Internet Key Exchange) packets. The IPsec service is stopped first so that the initial packets can be captured.

Question. (What parameters are negotiated during the IKE exchange?): While observing the negotiation, several parameters are mentioned: an integrity algorithm, pseudo-random function, and the Diffie-Hellman key exchange. These different values can be seen in the payload packed

Task 4 Capturing ESP:

Question. (What differences do you notice between the captured ESP packets and the plaintext packets from Task 2?): Observing the packets from Task 2 that are in plaintext, and then the packets that are encapsulated inside an ESP packet, the information is encrypted and scrambled.

Question. (Why is the payload data not visible in the ESP packet? (put screenshots on your report to show that)):

The payload data is not visible in the ESP packet because IPsec's Encapsulating Security Payload (ESP) protocol encrypts it.

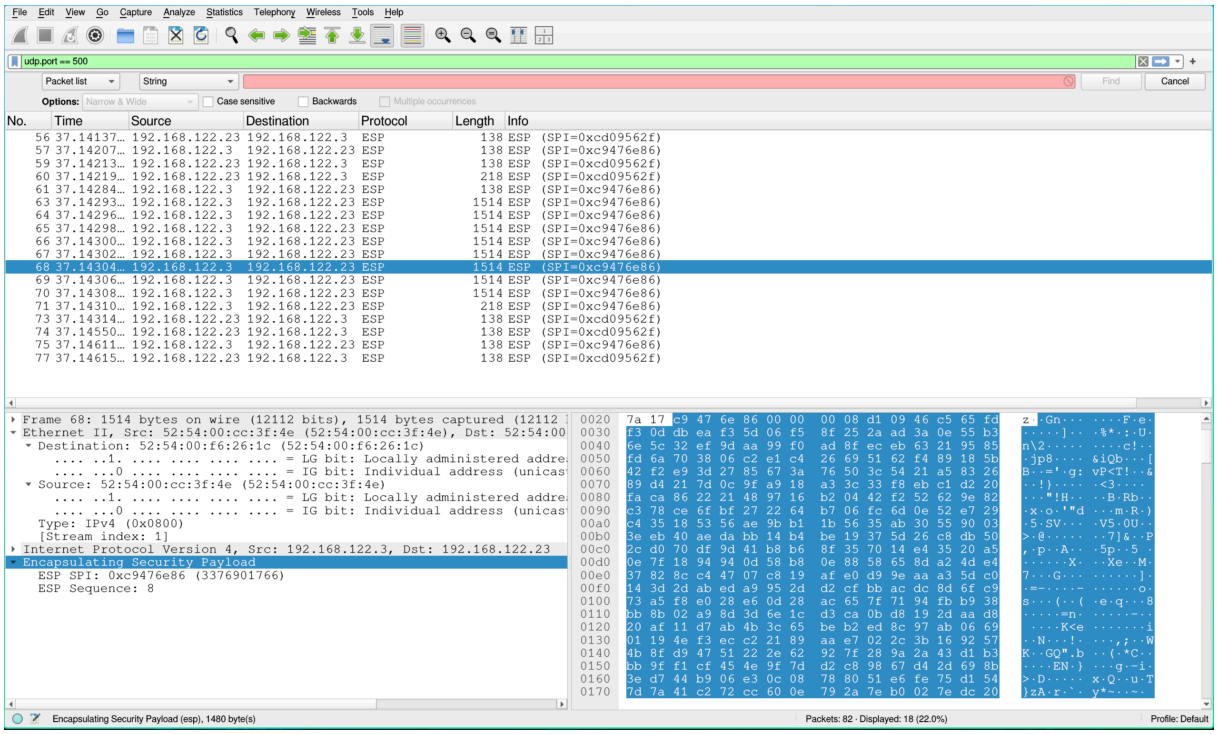

Figure 4: ESP traffic

As seen in the Figure 4 is the is the screen shot of the ESP filter

2.2 Assignment VPN part two

SSL/TLS VPNs vs IPsec:

SSL/TLS VPNs are a method to establish a VPN connection over the TLS protocol. They use the HTTPS protocol to communicate and encrypt data. The way it works is that the client's packets are encapsulated inside TLS encryption and sent to the VPN server. The VPN server decrypts the packets and forwards the traffic to the final destination on behalf of the client. The response from the destination server is then returned to the VPN server, which re-encapsulates it in TLS and sends it back to the client. Since SSL/TLS VPNs operate over HTTPS, they are firewall-friendly. The SSL/TLS VPN protocol operates at the application layer. Comparing this protocol with IPsec. IPsec operates at the network layer, and therefore the protocol needs to establish a key-exchange method. There are two main methods: Internet Key Exchange (IKEv1) and Internet Key Exchange version 2 (IKEv2). Compared to IPsec, SSL/TLS VPNs are more effective at bypassing normal firewalls, since IPsec traffic can sometimes be blocked or require extra configuration.

WireGuard vs IPsec:

The WireGuard is a more modern VPN. It uses the following protocols:

- ChaCha20 for symmetric encryption, authenticated with Poly1305,
- Curve25519 for key exchange,
- SipHash24,
- BLAKE2s for hashing,
- HKDF for key derivation.

One of the features that WireGuard is primarily designed for is its integration in the Linux kernel, which makes installation and setup easy. WireGuard uses Curve25519 to derive the key-exchange method. Another technique that WireGuard uses is frequent rotation of the session keys, which makes the protocol more secure while still maintaining the fast connection that is one of the key features of WireGuard.

To compare this protocol to IPsec: both operate in the same network stack at Layer 3, but WireGuard has a much smaller code base, whereas IPsec has a much larger code base that makes IPsec more configurable and able to run on most operating systems. This lean design also means WireGuard is easier to audit and maintain, reducing the potential attack surface compared to the more complex IPsec implementation. While IPsec supports a wide range of cipher suites and authentication methods, which contributes to its flexibility, this complexity can also lead to more configuration errors and higher administrative overhead. WireGuard, by contrast, focuses on a fixed set of modern cryptographic primitives, providing strong security with minimal configuration and typically faster connection setup.

3 Transport Layer Security (TLS) + Secure Shell (SSH)

3.1 Assignment TLS Cipher Suite (Individual) + Analyze their components

Review Valid Combinations of TLS Cipher Suites mentioned in the slides

- **Assignment** *'Study the provided list of valid TLS cipher suites'*:

1) **Key Exchange**
   In the 10 valid TLS cipher suites, three different key exchange methods are used: *RSA, DHE, ECDHE*. RSA is a method where the two parties use public keys to encrypt the symmetric key that both sides will use. DHE (Diffie-Hellman Ephemeral) is a technique where the two parties use the Diffie-Hellman algorithm to derive a shared secret key that will be used to encrypt the messages they send to each other. ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) is another method to derive a shared secret key, but this one uses elliptic curve cryptography to achieve stronger security with smaller key sizes.
2) **Authentication**

   RSA is a public-key algorithms. That can be used for authentication of the server, by signing secure digital messages and certificates. The security of RSA relies on the

   ECDSA (Elliptic Curve Digital Signature Algorithm) works in a similar to the RSA way but is based on elliptic-curve cryptography.
3) **Encryption Algorithm + Mode**

   AES is a symmetric-key encryption algorithm and is the one used most often in the valid TLS cipher suites from the slides. AES is a block cipher that can use three key lengths: 128-bit, 192-bit, or 256-bit. The mode defines how the blocks of data are processed. Common modes in TLS include CBC (Cipher Block Chaining) and GCM (Galois/Counter Mode). CBC mode encrypts each block based on the previous one, while GCM mode provides both encryption and built-in integrity verification.
4) **MAC Function**
   Message Authentication is a method used to verify that a message truly comes from the sender the client is communicating with. Integrity ensures that the parties in a communication channel can confirm that none of the messages have been tampered with during transmission. In TLS, this is achieved using a MAC function such as SHA256 or SHA384, which creates a unique fingerprint for each message. If the message changes in any way, the fingerprint no longer matches, and the receiver can detect that the data has been altered.

Design 3 "Impossible" Cipher Suites + Justify Each Invalid Combination :

1) **TLS_DH_DSA_WITH_AES_128_CBC_SHA:**
   This combination is invalid because the Diffie-Hellman (DH) and the DSA are not compatible for key exchange and authentication, since DSA is designed only to sign messages. Therefore, there is no authentication in this cipher suite.
2) **TLS_AES_RSA_WITH_ECDH_GCM_SHA256:**
   AES is a symmetric encryption algorithm and cannot be used for key exchange, while ECDH is an asymmetric key exchange method. Combining them in this order makes the suite structure incorrect and therefore an impossible combination in TLS.

3.2 Assignment SSH MITM attack (Individual)

First, to perform the ARP spoofing attack, the Man-in-the-Middle attack has to be prepared too, so that Ettercap GUI can be used to set the two targets and then start the Man-in-the-Middle.

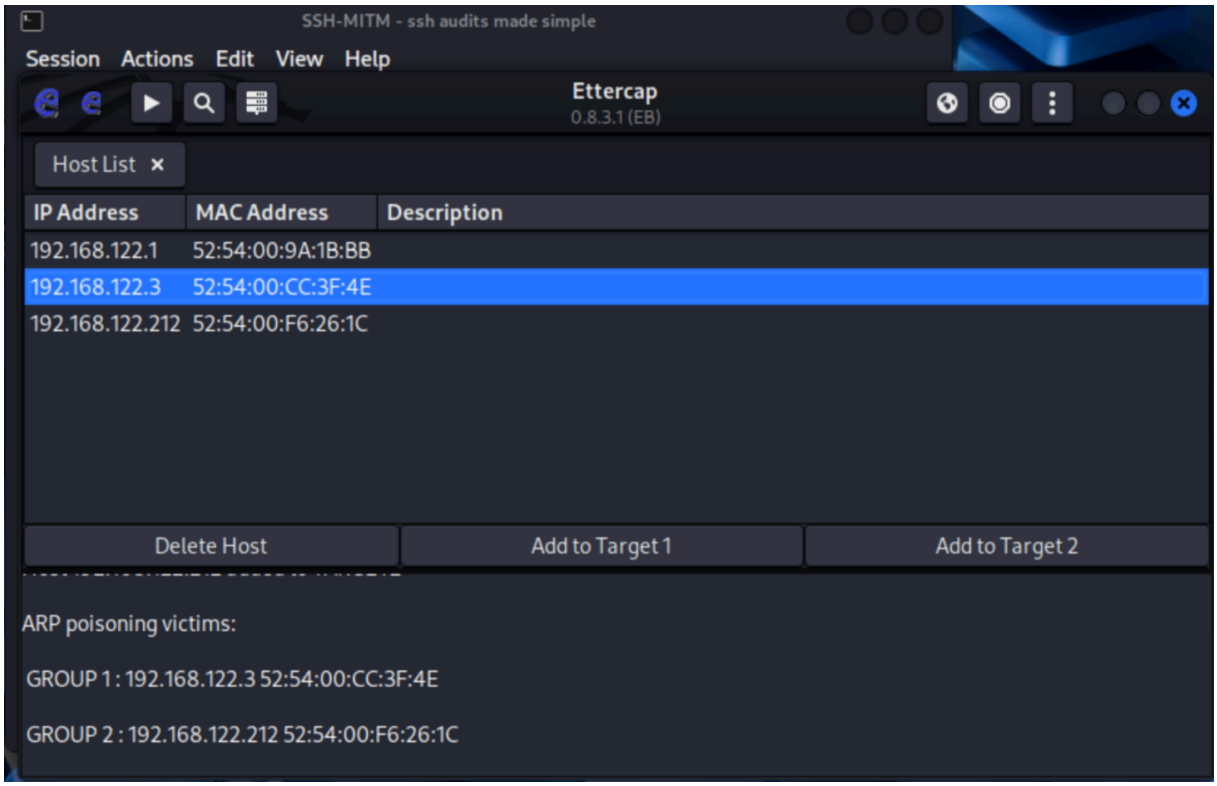

Figure 5: Ettercap GUI starts the MITM

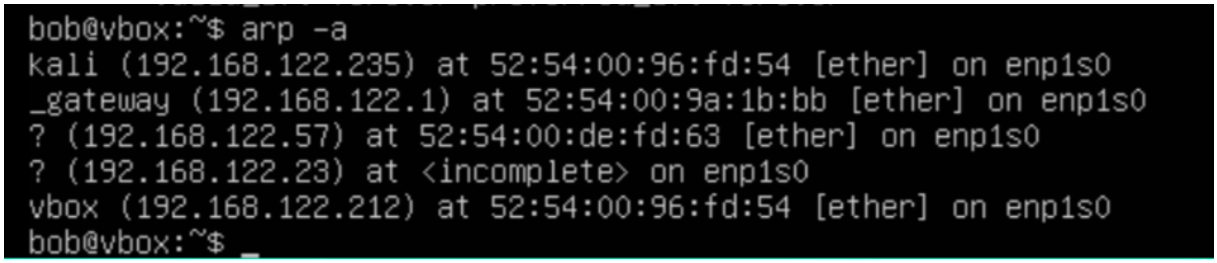

Figure 6: Bob the client

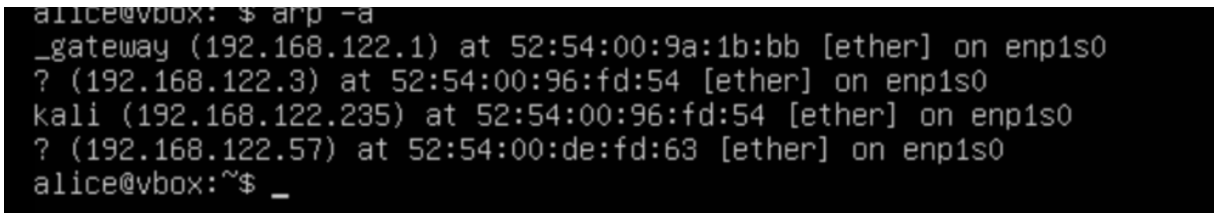

Figure 7: Alice the server

As Figure 6 and Figure 7 show, the server and the client have changed the MAC address to the Kali machine's MAC.

Now the Man-in-the-Middle SSH can be performed with the following command:

```
$ ssh-mitm server --remote-host 192.168.122.212
```

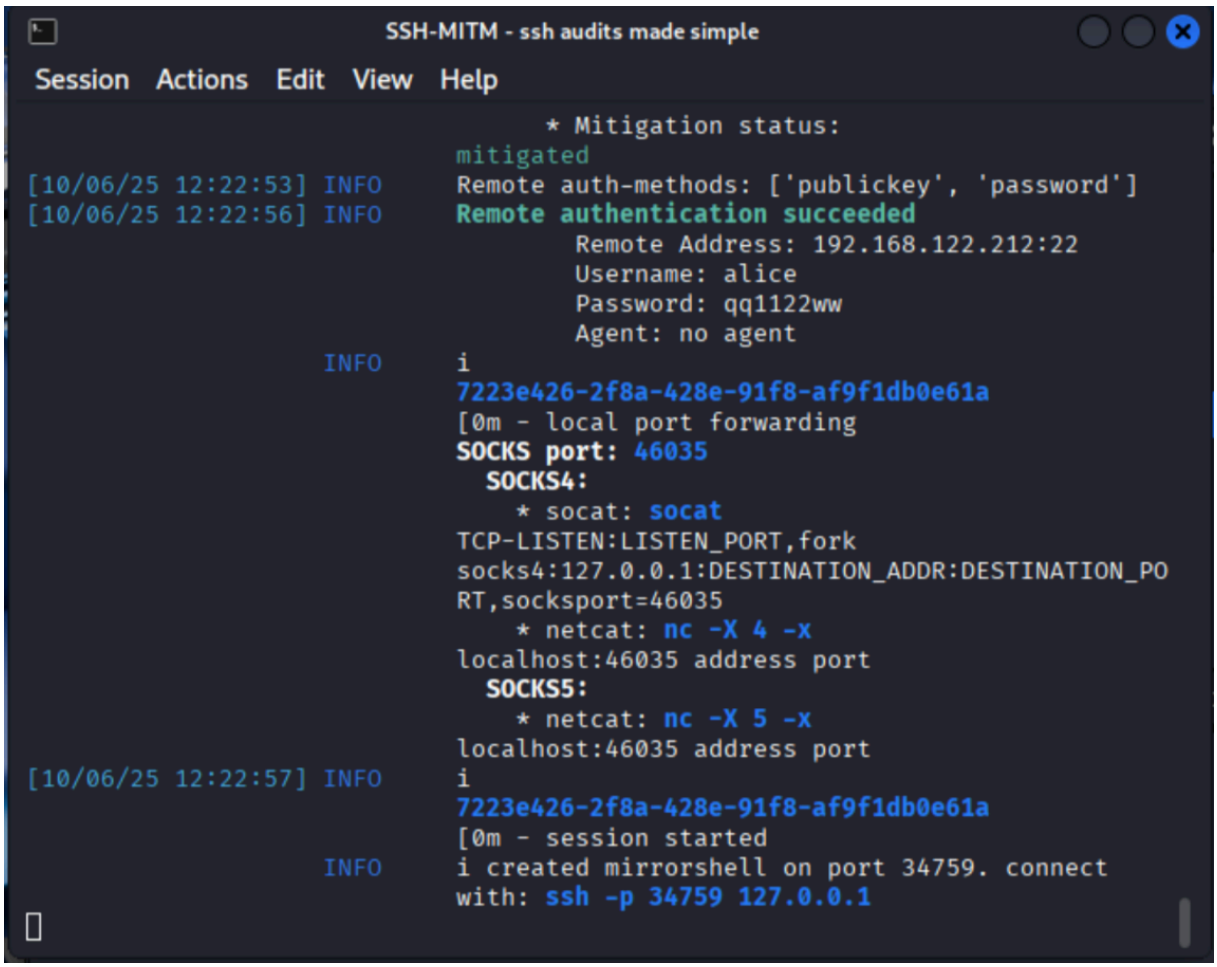

Figure 8: ssh-mitm

After the successful attack, 'ssh-mitm' showed some CVEs. I think these CVEs are vulnerabilities that can be exploited to make a more persistent attack. I also noticed that when I tried to connect to Alice (the server) again, I got an SSH warning saying "there's an eavesdropper, possibly a Man-in-the-Middle attack".

Exploring the hijack, I saw that I could enter the SSH session and start typing and using the shell that Bob had just started. And the typing was also showed on the Bob machine.

4 WiFi security

4.1 Assignment WiFi (Group)

I did the group assignment, but when we wanted to deauth with WiFi pineapple. So I decided to do the assignment individually, without the WiFi Pineapple, since I have my own internet adapter that can do monitor mode, I can still do the assignment.

To set up for the assignment, I used a Raspberry Pi to create a hotspot. The commands I used were:

```
$ sudo nmtui
```

After setting up the hotspot, I connected to the Raspberry Pi with my laptop and mobile phone to start traffic and start pinging each other. While the traffic was running, I started the ALFA adapter in monitor mode and started airodump-ng to capture the traffic, with the following commands:

```
$ sudo ifconfig wlan1 down
$ sudo iwconfig wlan1 mode monitor
$ sudo ifconfig wlan1 up
# to check which channel the access point is on
$ sudo airodump-ng wlan1 -c <channel>
```

Now while the traffic was running, the next step is to deauth the connected clients from the hotspot to capture the handshake.

```
$ sudo airodump-ng --bssid <bssid> -c <channel> -w capture wlan1
$ sudo aireplay-ng --deauth 100 -a C6:60:AD:1A:5E:38  wlan1
```

```
CH  8 ][ Elapsed: 1 min ][ 2025-10-14 11:05 ][ WPA handshake: C6:60:AD:1A:5E:38

BSSID              PWR RXQ  Beacons    #Data, #/s  CH   MB   ENC CIPHER  AUTH ESSID

C6:60:AD:1A:5E:38  -38 100      956      143    0   8  180   WPA2 CCMP   PSK  SanderPhone

BSSID              STATION            PWR   Rate    Lost    Frames  Notes  Probes

C6:60:AD:1A:5E:38  70:1A:B8:C5:D3:F2  -27    1e- 1e     0      182  EAPOL
```

Figure 9: Captured handshake

In Figure 9, shows that the handshake is captured and now the next step is to try to crack the password with the tool John the Ripper. But before cracking the password, the capture file have to be converted to a hash file that John can accept.

```
$ wpapcap2john SanderHand-01.cap > SanderHandJohon.john
$ john  SanderHandJohon.john
```

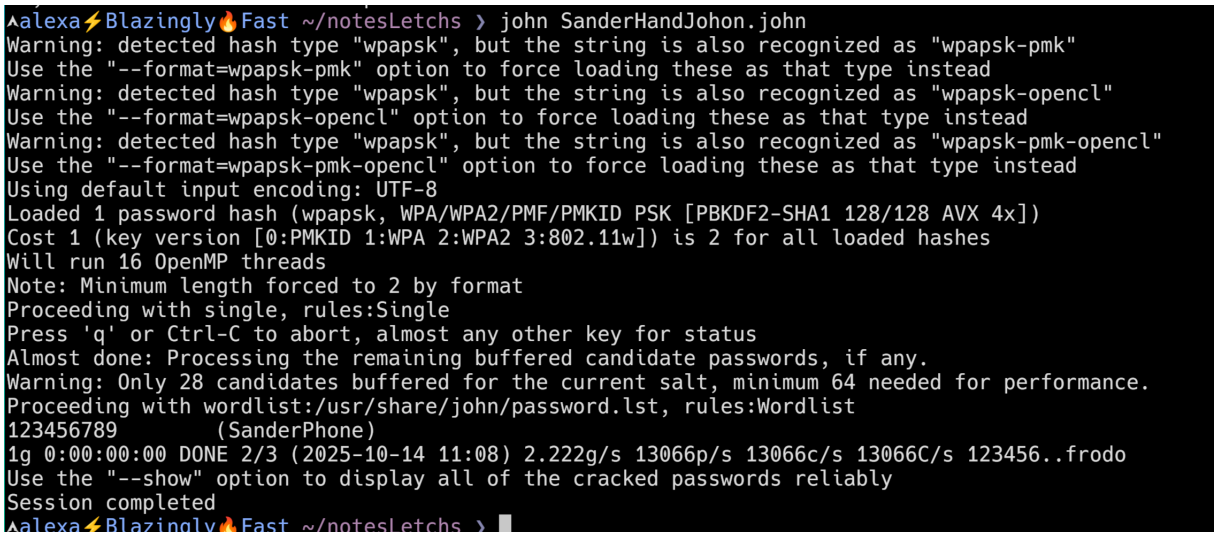

Figure 10: Captured handshake

As Figure 10 shows, the password is cracked and it took less then 1 second to crack the password, since the password is a weak password. This is why its important to always have a strong password on your WiFi access point, so that attackers not can easily crack the password and get access to your private network.

4.2 Assignment Wi-Fi Attack Names

Man-in-the-middle attacks

This is an attack where the attacker places themselves between two parties that are communicating. The attacker can then intercept, modify, block the communication, or simply just listen to it. The attacker can perform an **SSL hijack** attack that downgrades the HTTPS protocol to HTTP, allowing them to see all communication in plain text. For example, the attacker downgrades the HTTPS login page to HTTP, and now they can see the username and password being sent.

DNS spoofing is a technique where the attacker has established a man-in-the-middle attack, and can then intercept DNS requests and respond with a malicious IP address. This allows them to redirect the user to a fake website — for instance, a fake Facebook login page. Even if there is an OTP (one-time password), the attacker can still design the website to look identical to the real Facebook page. When the user enters their username and password, the attacker can then perform an account takeover. This is why it is important to notice the when the bar that the URL is not correct and the URL bar is not saying: "Not secure", and there is a lock icon with a red cross on it.

Network injection — router / access point compromise

An attacker may target a Wi-Fi router or access point by exploiting firmware vulnerabilities, weak/default credentials, or try to inject malicious code to execute unauthorized commands gain access. Once the attacker gains administrative access they can modify firmware logic or configuration so the device behaves according to the attacker's wishes: capturing and logging all passing traffic, altering DNS responses or creating persistent backdoors for later access.

Deauthentication attacks are a type of attack where the attacker sends deauthentication frames to the target device, forcing it to disconnect and after the disconnection, the target device will try to reconnect to the access point, with a WPA handshake. The attacker can then capture the handshake and try to crack the password offline.

5 Cryptographic Key Management and Key Distribution + X.509 Certificates

5.1 Assignment: Trusted Root Certificates

The two trusted root certificates I have chosen are from GoDaddy and Microsoft.

GoDaddy:

```
CN = Go Daddy Root Certificate Authority - G2
O = GoDaddy.com, Inc.
L = Scottsdale
S = Arizona
C = US
```

Microsoft:

```
CN = Microsoft RSA Root Certificate Authority 2017
O = Microsoft Corporation
C = US
```

1) **Who (which entity) has signed this certificate?**
   Both the GoDaddy and Microsoft root certificates are self-signed. This means that these are root certificates and therefore sit at the top of the hierarchy. They are the only ones that have signed their own certificates.
2) **Why is it trusted?**
   The two certificates are trusted because they are pre-installed in the operating system, allowing the OS or the browser recognize and trust these certificates automatically.

5.2 Assignment: What's My Chain Cert?

I deciested to pick my own domain:
softrunner.dk

1) **How many certificates do you see in the chain?**
   There is four certificates in the chain
2) 1) **For each certificate**
      Server (End-Entity) Certificate
      Subject: `CN=softrunner.dk`
      Issuer: `C=US, O=Google Trust Services, CN=WR1`
      Role: Server Certificate
      - This is the website certificate issued to `softrunner.dk`.
   2) **Intermediate Certificate**
      Subject: `C=US, O=Google Trust Services, CN=WR1`
      Issuer: `C=US, O=Google Trust Services LLC, CN=GTS Root R1`
      Role: Intermediate CA
   3) **Root CA Certificate**
      Subject: `C=US, O=Google Trust Services LLC, CN=GTS Root R1`
      Issuer: `C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA`
      Role: Root CA (Sub-root)
   4) **Global Root Certificate**
      Subject: `C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA`
      Issuer: Self-signed (same as Subject)
      Role: Trusted Root CA
3) **Chain format**:
   softrunner.dk (CN=softrunner.dk) → Google Trust Service (CN=WR1) → GTS Root R1 (CN=GTS Root R1) → GlobalSign Root CA (CN=GlobalSign Root CA)
4) **Which certificate in the chain expires first? And why are end-entity certificates usually valid for a much shorter period (e.g., 90 days)?**

   The certificate will expire first is the server (end-entity) it will expire on the **Jan 16 01:34:01 2026 GMT**. The end-entity is usually valid for a much shorter period because shorter lifetimes reduce security risks in case the private key is compromised or misused.
5) **Compare the signature algorithms (e.g., SHA256WithRSAEncryption).**
   Most of the certificates in the chain use the **sha256WithRSAEncryption** algorithm, except for the root certificate from GlobalSign, that uses **sha1WithRSAEncryption**. It might be because the root is signed in the **Sep 1 12:00:00 1998 GMT**, so maybe at its was common to use SHA-1 back then.

5.3 Assignment: Explore Certificate Transparency (CT) Logs

I decided to explore two domains: aau.dk and mitid.dk but i used the site: https://crt.sh/ and the https://platform.censys.io/:

Domain: aau.dk **CA Issued**: Let's Encrypt
Number of certificates: Counting only the Common Name (CN) "aau.dk" and excluding subdomains, there were around 40 certificates.

Duplicates or expired certificates: Let's Encrypt issues both a leaf certificate and a precertificate.

Most recent certificate logged: The most recent certificate was issued on 2025-09-18.

Domain: mitid.dk
CA Issued: Let's Encrypt
Number of certificates: Counting only the Common Name (CN) "mitid.dk" and excluding subdomains, there were around 23 certificates.

Duplicates or expired certificates: Let's Encrypt issues both a leaf certificate and a precertificate, so some entries appear as duplicates.

Most recent certificate logged: The most recent certificate was issued on 2025-09-03.

Why are CT (Certificate Transparency) logs important? CT logs are important because website and peple can monitor and see the history of certificates issued for their domains. This helps detect malicious or misissued certificates, and can enhance the security by making the certificate transparent.

How could CT logs help detect mis-issued or rogue certificates?

Who maintains these logs, and how can they be trusted themselves?
CT logs are maintained by independent organizations (for example, Google, Cloudflare, DigiCert, Let's Encrypt, and others).

Can you find a phishing domain which uses a certificate?

- **Find some information about that certificate such as validity period etc. Do you see any differences between this certificate and a certificate from a valid domain (like aau.dk etc.)**

I don't have more searches on the https://platform.censys.io/ so i can't preform this task

6 Digital Signature & Bitcoin

6.1 RSA challenges for signing

Objective:

Identify and explain at least three specific challenges of RSA that DSA aimed to address

1) DSA is designed specifically for digital signatures, to validate where the RSA algorithm can encrypt, sign, and perform key exchange. This means that DSA has been implemented so when creating a signature, four components are used: the message, the private key, a random value, and the hash function. This will create a unique signature for this message. To verify the signature, the client that receives the message can use the public key and the message to verify whether the signature is valid or not. This will return true or false.
2) DSA will always produce different signatures even if the same message is signed multiple times with the same private key. this occurs because DSA uses a random value (nonce) during the signing process, ensuring that each signature is unique.
3) DSA dont use the encryption like RSA does, this means that DSA is With RSA, the same algorithm is used for encryption and signing.

6.2 Digital Signatures in practice

Objective:

The objective of this assignment is to help you understand how different digital signature algorithms work by generating and comparing signatures for the same text using Python.
To find the python implementation see the appendix section.
to run the code, use the following command in the terminal:

```
$ python3 createsigns.py "Your message here"
```

```
alexa Blazingly Fast ~\..\SLP main > py .\morduls\m6sig.py "HELLO WORLD"
Imported keys

About to generate signature for RSA
Signature took 6485400 ns (0.0064854 s)
RSA signature: 4f9a7611c8d58393ba5c0b557ff318995f2840483b42611ca2019ec897dfabc56b411ddf5ffa956
911aaa6d8a9f56f6047cf5a216a7d770541e6af2cff68a8a8b978117a023c49c03d950afad119e76dc784832e25084
94ecf3df1e8e9d94f0a2793d664445e8f4684b5f9200c56df51f46f138a2052af65eb9f7d89a27c3c189c78eaaec50
9db3ca767446fd3011b5fd439361e71789f2d81184dae8695899c8456ec1a591045e38ce98097ff8565d9c3f382fa1
c9bd2417b593892f992005b7bcae184d430ca82d83dfe26e05d77b8c3235930c411f65fdd383e4d490c006dd18718d
4936b36831f8a095b74c79db873eea2afb318521c8d491cead7a851ad

About to generate signature for DSA
Signature took 1031200 ns (0.0010312 s)
DSA signature: fa3e7549cf8566ef64a0a1c62092053c58f565183e1921953de372a9d465641be503f1840e2b8f8
c6212b384881a923b43bf32d1d7220e51

About to generate signature for ECDSA
Signature took 999800 ns (0.0009998 s)
ECDSA signature: 99a270a3626a8be34f58da6d7d3cc1d791c7db92ca6b2a2791995c3843f33a8b52f633c752e57
dbf56c9b3ce9af8a1d9bb8f41077a2c2283754bd679cae00e48

alexa Blazingly Fast ~\..\SLP main >
```

Figure 11: The three signatures generated using RSA, DSA, and ECDSA

As seen in Figure 11, three different signatures were generated using RSA, DSA, and ECDSA. Notice that each signature has a different length and structure, this is due to the distinct mathematical algorithms. An interesting observation is that when running the script with the same input message, the signatures produced by DSA and ECDSA change every time. This happens because both algorithms include a random value (nonce) during the signing process,

ensuring that each signature is unique even when the same message and key are used. In contrast, the RSA signature remains identical for the same input message, as it does not rely on randomness once the key pair is fixed. The performance of RSA algorithm takes noticeably longer, around four times slower than DSA. This is because it involves more computationally intensive operations with large modular exponentiations. DSA and especially ECDSA are designed to be more efficient, with ECDSA achieving by using much smaller key sizes.

Regarding signature length, RSA produces the longest signatures, while ECDSA signatures are the shortest. This is because elliptic curve cryptography provides equivalent security with smaller mathematical parameters

6.3 Dual Signatures

– Task 2: **Secure Payment Design**:

Creation of dual signatures: The generation and implementation of the SET protocol for dual signatures is done by first generating two hashes — one for the payment information and one for the order information. These hashes are then saved so they can be verified later. After generating the two hashes, the generation of the POMD hash can be done by combining the two other hashes See the code below:

```
PI = b"Pay 100 DKK to MerchantX"
OI = b"Order #2456: 2 items total 100 DKK"

with open('PI.txt', 'wb') as fp:
    fp.write(PI)
with open('OI.txt', 'wb') as fp:
    fp.write(OI)

PIMD = SHA256.new(PI)
OIMD = SHA256.new(OI)

with open('PIMD.bin', 'wb') as fp:
    fp.write(PIMD.digest())
with open('OIMD.bin', 'wb') as fp:
    fp.write(OIMD.digest())

POMD = SHA256.new(PIMD.digest() + OIMD.digest())
```

Once this hash is created, the RSA key can be generated. The values are stored in two files: key.pem as the private key and pubkey.pem as the public key. After the key has been generated, we can use the private key to sign the POMD.

```
sig = pss.new(key).sign(POMD)

with open('sig.bin', 'wb') as fp:
    fp.write(sig)
```

Os this means that the key is generated first based on the order and payment information that have been hashed, and then the two digit are combined and hashed again. This final hash is the POMD, and this POMD is then signed. The resulting signature is saved in the file sig.bin.

Verification of dual signatures:
To verify the dual signature, the process starts by loading the public key from the pubkey.pem file. If we want to verify the signature from the Merchant perspective, we need two hashes: let's say we have OI and PIMD. We hash the OI to generate the OIMD and then combine this with the PIMD to generate the POMD. After generating the POMD, we load the signature from the sig.bin file. Then, we use the public key to verify the signature against the POMD.

```
with open('OI.txt', 'rb') as fp:
    OI = fp.read()
with open('PIMD.bin', 'rb') as fp:
    PIMD = fp.read()
OIMD = SHA256.new(OI)
POMD = SHA256.new(PIMD + OIMD.digest())
try:
    verifier.verify(POMD, sig)
    print("Merchant verified signature")
except (ValueError):
    print("Merchant cannot verify signature")
```

To verify the signature from the Bank perspective, the process is similar. Here we already have the OIMD, and we can generate the PIMD by hashing the payment information. Then, we combine the OIMD and PIMD to obtain the POMD. After generating the POMD, we load the signature and verify it using the public key.

Screenshots of generated signatures and verification results:

Generation:

```
alexa Blazingly Fast ~\..\setproto main > py .\create.py
PI=b'Pay 100 DKK to MerchantX'
OI=b'Order #2456: 2 items total 100 DKK'
PIMD.hexdigest()='42b1efdb293e0221302d7181fcbaf031bded3eaec99d0547a6b57507e3a72cfd'
OIMD.hexdigest()='513d07a55b21f3fc4d4d4855c132b7449a1f01406136c178104c1e18bf559de4'
POMD.hexdigest()='38993878f29498879aadb9e497f10d8d2c53ad99296cd7667b8330bd2e3f1099'
sig.hex()='0871138c186a8433f2557fc6460a250c1084fe64ea41a993c7ce538fab4d593198915b5567
cd7b463b39544c211a4e4fa936e7d68f381e477cee6af9217366ee930b6168c31c1f58987cc1dddbaa530
8357a4f6a7aa648f4e5961a11d9abd74452cbd2ad45a24504f57ef8ecb54b4c3dcd1b72f7b4bca88cc288
a8b1f6539d946c6800535ec34ec769be2018aa6eca0c39bc6522c8115a757df2e12a3e3e6843c066208b3
6445ac447ce3292b6b493114212ab145d0a3b5a765ed74f73d275175fe3985ceccd7c74256ffc7bd6fc53
e4a96a0037ff713b3b6e88cf2b3e9fbe8b1c1036797faf3813fb3b75f64cf142d1227f00c4b1054449431
78f7cf970237c'
alexa Blazingly Fast ~\..\setproto main >
```

Figure 12: The generated dual signature

Verification:

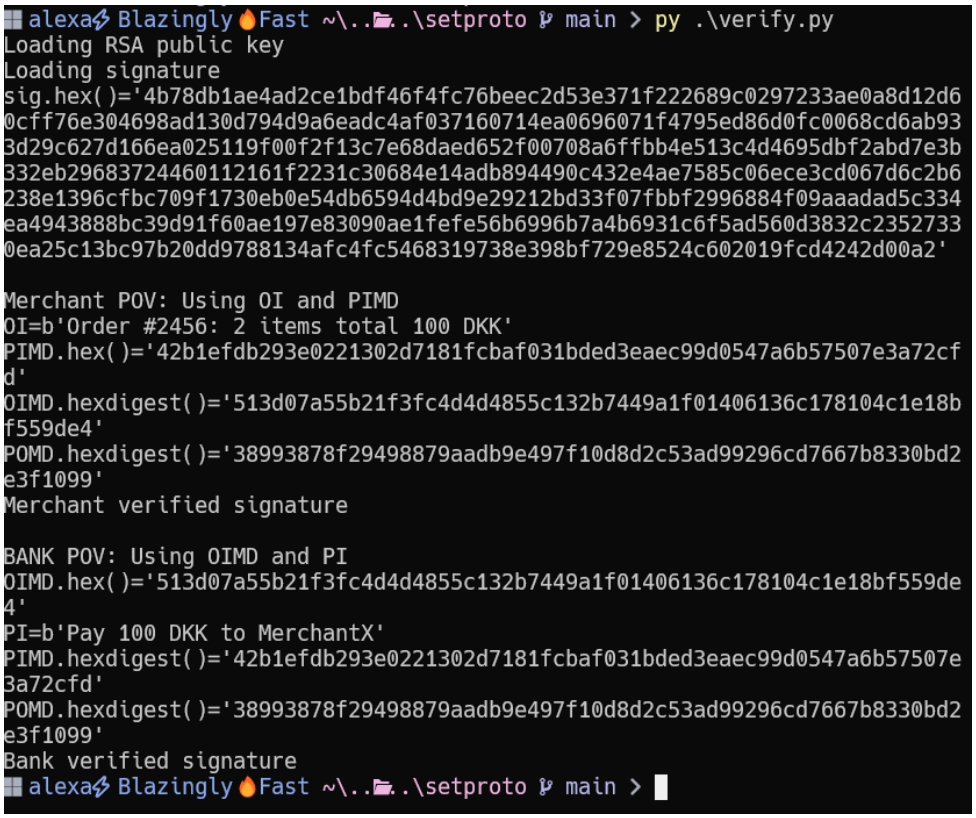

Figure 13: The verification of the dual signature

7 Electoinlick Mail seacrt

7.1 The State of Email Security

DNSSEC adoption in the real world: Why is DNSSEC not universal yet, and what are the main deployment barriers?

The DNSSEC protocol is used in the real world, but not all sites have adopted it. The protocol is complex, and since it is built on top of the old DNS protocol, it becomes even more complicated. Many people complain that DNSSEC is too complex to set up and get working properly. The main deployment barriers are the technical complexity, the need for careful key management, and the risk of misconfiguration that can take a domain offline. Many organizations also find it difficult to maintain because DNSSEC requires extra setup and regular updates to keep the keys valid.

How Proton Mail work, is it PGP or S/MIME? Why?

Proton is usning PGP (priti good pricy) the diginso of the PGP is that its ueses asymmetric encryption that uses pupclik and prive key as the why to preicert the email being sned, htis is only being done for hte userss that are suing a proton mail, event alle the messinges that are sored in the server of proin is encruede so this is only the user and tha can see the emils being resived and resnd. this conset is now as zero-access encryption.

Tuta vs Proton Mail

As we know, Proton Mail uses PGP, but the other service, Tuta, does not adopt the PGP design. This is because PGP is not considered quantum-resistant. Tuta instead uses the Kyber-1024 algorithm, which is a quantum-resistant encryption algorithm. They clearly state that they are not using the PGP design since it cannot be easily adapted to quantum protection. On the other hand, the pricing of the two services is similar — both offer a free plan, with around 1 GB of storage, and paid plans costing about 3 dollars for around 15–20 GB of storage

7.2 Verifying the Email Security Stack of mitid.dk

Step 1) Verify DNSSEC:

To retrieve the **RRSIG**, the bash command shown below will get this signature and the result:

```
$ dig @1.1.1.1 +dnssec mitid.dk
mitid.dk. 20 IN RRSIG A 13 2 20 20251105204733 20251102194733 65041 mitid.dk. /
L0NsAx6Un/hM5nfB9hWVZr4EJ5HTRl/R+YQ6uPUWyRRG+OsiDTOBdLV 1dVX1mvSo/...
```

```
nssec mitid.dk

; <<>> DiG 9.20.13 <<>> @1.1.1.1 +dnssec mitid.dk
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37528
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 1232
;; QUESTION SECTION:
;mitid.dk.                      IN      A

;; ANSWER SECTION:
mitid.dk.               20      IN      A       95.100.155.16
mitid.dk.               20      IN      A       95.100.155.160
mitid.dk.               20      IN      RRSIG   A 13 2 20 20251106084733 20251103074733
 65041 mitid.dk. 5CnWH5RejkZBCD0CE2+2Nb0FMIsV/BpU2Tz7fS+WjqvWgVH6W96ZoJmK N+ElVb2oB//X9
ImUS1ljaWAGb8ScYA==

;; Query time: 10 msec
;; SERVER: 1.1.1.1#53(1.1.1.1) (UDP)
;; WHEN: Mon Nov 03 17:38:39 CET 2025
;; MSG SIZE  rcvd: 173
```

Figure 14: A capture of a dig command

As seen in Figure 14 , this is the response showing both the A record and its corresponding RRSIG.

The (RRSIG A) is the signature for the **A record**, which means the IPv4 record. (13) is the algorithm used for this signature — in this case, it's using **ECDSA Curve P-256 with SHA-256** since its number is 13. (2) is the number of labels in the record name. (20) is the TTL. The next two numbers are the timestamps — the first is the **expiration date** (2025-11-05 20:47:33), and the second is the **signing (inception) date**. (65014) is the **Key Tag** that should verify the signature. After mitid.dk is the **digital signature** that has been hashed using SHA-256. This key is encoded with **Base64**.

The next command is to find the **DNSKEY**:

```
$ dig @1.1.1.1 +dnssec DNSKEY mitid.dk | grep mitid.dk | cut -c1-100
mitid.dk.   5534    IN      DNSKEY  256 3 13 pRrLFT7evmDj1V2n/....
mitid.dk.   5534    IN      DNSKEY  256 3 13 8WioCwcmhPA2YSyt/....
mitid.dk.   5534    IN      DNSKEY  257 3 13 Zp1SZuFNXD53OU7K/....
mitid.dk.   5534    IN      DNSKEY  257 3 13 8gcnnjVQPkfmZxU+/....
mitid.dk.   5534    IN      RRSIG   DNSKEY 13 2 7200 20251106084733 20251103074733
3949 mitid.dk. axtWTuBQENcW...
mitid.dk.   5534    IN      RRSIG   DNSKEY 13 2 7200 20251106084733 20251103074733
44941 mitid.dk. JxDw7V91O4R...
```

There are four DNSKEYs — two of them are **ZSKs** and the other two are **KSKs**. The **ZSK (Zone Signing Key)** is used for signing records like MX, A, etc. The **KSK (Key Signing Key)** is used for signing the DNSKEY RRset itself. The flag value 256 identifies a ZSK, and 257 identifies a KSK. The 3 is the protocol number for DNSKEY. The 13 is again the algorithm **ECDSA Curve P-256 with SHA-256**. Next comes the **public key**, which is used to verify the signatures in the RRSIG. Here is where the **Key Tag** comes into play, since this value is used

to locate the signer. The RRSIG DNSKEY is a different signature — it signs the **DNSKEY RRset**, allowing the DNSKEYs themselves to be verified.

The next part is understanding the **DS record**:

```
$ dig @1.1.1.1 +dnssec DS mitid.dk
mitid.dk.     6060    IN      DS      44941 13 2 6DA9443050311FF6DE97629...
mitid.dk.     6060    IN      RRSIG   DS 13 2 7200 20251129104202 20251101091202 50934
dk. qjmeMFyQSUdjY0Y0fk...
```

The **DS record** is the record stored in the **parent zone (root or .dk)** that refers to the child's DNSKEY. Here, 13 is again the algorithm **ECDSA Curve P-256 with SHA-256**. The DS record contains a **hash (digest)** of the child zone's **public key (DNSKEY)**, which is used to validate that key. There is also an **RRSIG for the DS**, which is the signature from the parent zone proving that the DS record itself is authentic.

1) **Which one represents the digital signature?**
   the RSSIG is the one that represents that digital signrueture
2) **which one stores the public key?**
   The DNSKEY is the one that sores the pubkey key
3) **What are the 'flags' in the header? Explain each of the flags**

   ```
   $ dig @1.1.1.1 +dnssec mitid.dk
   ;; flags: qr rd ra ad; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1
   ```

   The flags explained:
   qr – this indicates whether it's a query or a response. If the bit is 0, it's a query; if it's 1, it's a response. **rd** – Recursion Desired, meaning that recursion is allowed. **ra** – Recursion Available, showing that the server supports recursive queries. **ad** – Authenticated Data, meaning the DNSSEC signatures have been validated successfully.

How can you tell that DNSSEC validation succeeds for mitid.dk?

to valute the mitid.dk i can use the delv command to validate the tool chain

```
$ delv @1.1.1.1 +dnssec +rtrace +vtrace  mitid.dk
;; fetch: mitid.dk/A
;; validating mitid.dk/A: starting
;; validating mitid.dk/A: attempting positive response validation
;; fetch: mitid.dk/DNSKEY
;; validating mitid.dk/DNSKEY: starting
;; validating mitid.dk/DNSKEY: attempting positive response validation
;; fetch: mitid.dk/DS
;; validating mitid.dk/DS: starting
;; validating mitid.dk/DS: attempting positive response validation
;; fetch: dk/DNSKEY
;; validating dk/DNSKEY: starting
;; validating dk/DNSKEY: attempting positive response validation
;; fetch: dk/DS
;; validating dk/DS: starting
;; validating dk/DS: attempting positive response validation
;; fetch: ./DNSKEY
;; validating ./DNSKEY: starting
;; validating ./DNSKEY: attempting positive response validation
;; validating ./DNSKEY: verify rdataset (keyid=20326): success
;; validating ./DNSKEY: marking as secure (DS)
;; validating dk/DS: in fetch_callback_dnskey
;; validating dk/DS: keyset with trust secure
;; validating dk/DS: resuming validate
;; validating dk/DS: verify rdataset (keyid=61809): success
;; validating dk/DS: marking as secure, noqname proof not needed
;; validating dk/DNSKEY: in fetch_callback_ds
;; validating dk/DNSKEY: dsset with trust secure
;; validating dk/DNSKEY: verify rdataset (keyid=20109): success
;; validating dk/DNSKEY: marking as secure (DS)
;; validating mitid.dk/DS: in fetch_callback_dnskey
;; validating mitid.dk/DS: keyset with trust secure
;; validating mitid.dk/DS: resuming validate
;; validating mitid.dk/DS: verify rdataset (keyid=50934): success
;; validating mitid.dk/DS: marking as secure, noqname proof not needed
;; validating mitid.dk/DNSKEY: in fetch_callback_ds
;; validating mitid.dk/DNSKEY: dsset with trust secure
;; validating mitid.dk/DNSKEY: verify rdataset (keyid=44941): success
;; validating mitid.dk/DNSKEY: marking as secure (DS)
;; validating mitid.dk/A: in fetch_callback_dnskey
;; validating mitid.dk/A: keyset with trust secure
;; validating mitid.dk/A: resuming validate
;; validating mitid.dk/A: verify rdataset (keyid=65041): success
;; validating mitid.dk/A: marking as secure, noqname proof not needed
; fully validated
mitid.dk.               20      IN      A       95.100.155.16
mitid.dk.               20      IN      A       95.100.155.160
mitid.dk.               20      IN      RRSIG   A 13 2 20 20251106084733
20251103074733 65041 mitid.dk. sDcAOtXXEjRQFcIH77X6Bv+fGP/
SzYBgErRzBGxav32EKX6KNeDm5Du/ WUh+3jZwu/dzvzn/KAkwmaKIcZy2Uw==
```

As we can see, the chain of validation is shown above. It first starts with the A record for mitid.dk, which is then validated. After that, the DNSKEY for mitid.dk is fetched and validated, followed by the DS record for mitid.dk. Next, the DNSKEY for the parent zone (dk) is validated, and then the DS record for the parent zone, and finally the DNSKEY for the root zone (.). Observe how the order is resolved step by step — from the root, to the .dk zone, and finally to mitid.dk. This shows the complete validation chain moving from the top (root) all the way down to the A record. At the end, the message fully validated is shown. This means that all keys have been successfully verified — forming a complete, trusted chain of validation from the root zone down to the domain (mitid.dk), confirming that the DNSSEC validation succeeded

Step 2) Investigate SPF, DKIM, and DMARC

to request for the SPF the command as show blow can retireve the SPF flag:

```
$ dig @1.1.1.1 txt mitid.dkA
mitid.dk.               523     IN      TXT     "zf040yhy3q4ccv3vk9kv1fbwxb9wr1qp"
mitid.dk.               523     IN      TXT     "v=spf1 include:_spf.sitnet.dk
include:spf.protection.outlook.com ip4:91.102.26.64/27 ip4:185.208.80.30
ip4:185.208.80.157 ip4:91.102.30.2 ip4:91.102.30.3 ip4:91.102.30.130
ip4:91.102.30.131 ip4:91.102.30.140 ip4:188.64.157.0/29 -all"
```

As shown above, this is a list of all the IP addresses that can send email from the mitid.dk domain. There is also spf.protection.outlook.com, which is allowed to send email from this domain, and spf.sitnet.dk (this belongs to Statens IT, the State IT department of Denmark). I think this acts as a kind of backup - if all the other IPs are down, then MitID can still send emails through this domain or via spf.protection.outlook.com or spf.sitnet.dk

The "-all" means that all other hosts are not allowed to send email using the `@mitid.dk` domain.

DKIM

```
$ dig TXT default._domainkey.mitid.dk
```

this is no information on the flag

DMARC

```
$ dig TXT _dmarc.mitid.dk

;; ANSWER SECTION:
_dmarc.mitid.dk. 600 IN TXT "v=DMARC1; p=reject; sp=reject; rua=mailto:nets-
abuse@spam.riskiq.net;"
```

DMARC1 is version 1, where the p and sp tags mean that if the SPF check fails, the email will be rejected completely. This also applies to subdomains. The rua tag defines the email address where DMARC reports will be sent for analysis.

Are mitid.dk's emails protected from spoofing? Why or why not?

The short answer is yes, MitID is protected from spoofing. This is because the SPF record only allows specific IP addresses to send emails on behalf of mitid.dk, and the strict p=reject and sp=reject flags ensure that any email failing SPF or DKIM checks will be rejected.

Step 3) Check for DANE / TLSA

The posteo.de have the DANE TLSA:

```
$ dig TLSA _25._tcp.posteo.de
```

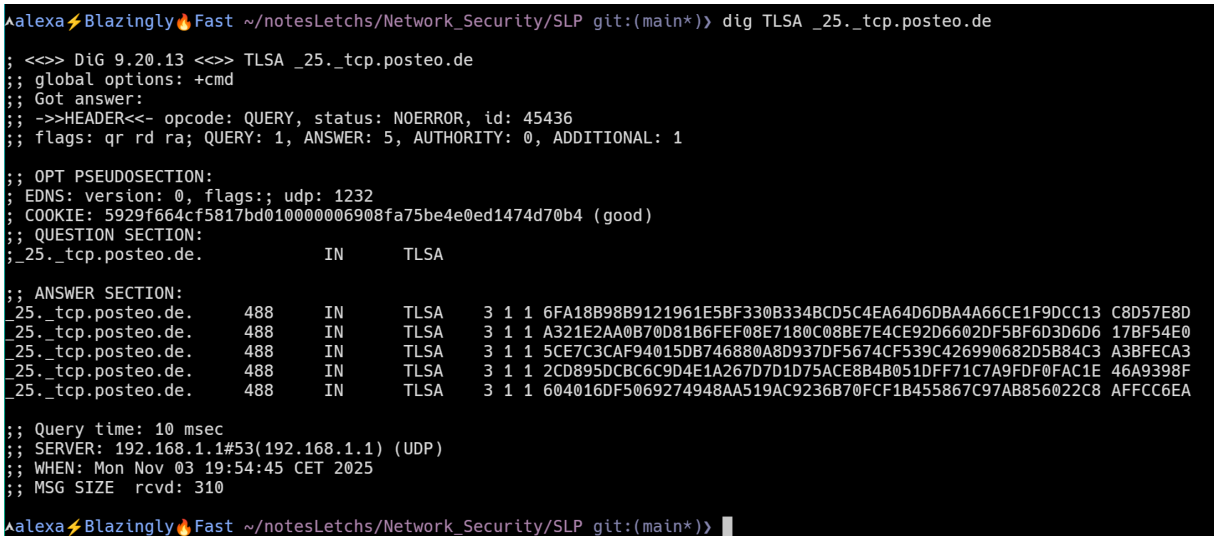

Figure 15: The screenshots of the dig commned

The reason for the screenshot is that it shows five answers, each with a fingerprint that is the hash of the public key.

8 Appendix section

8.1 Digital Signatures in practice in: 6.2

```python
#!/usr/bin/env python
from os import path
from time import time_ns
from Cryptodome.PublicKey import RSA, DSA, ECC
from Cryptodome.Hash import SHA256
from Cryptodome.Signature import pkcs1_15, DSS

def generate_keys():
    rsa_key = RSA.generate(2048)
    dsa_key = DSA.generate(2048)
    ecc_key = ECC.generate(curve='NIST P-256')
    return rsa_key, dsa_key, ecc_key

def export_keys(rsa_key, dsa_key, ecc_key):
    with open("rsa.pem", "wb") as fp:
        fp.write(rsa_key.export_key())
    with open("dsa.pem", "wb") as fp:
        fp.write(dsa_key.export_key())
    with open("ecc.pem", "wb") as fp:
        fp.write(ecc_key.export_key(format='PEM').encode())

def rsa_sign(data: str, key: RSA.RsaKey) -> bytes:
    h = SHA256.new(data.encode())
    return pkcs1_15.new(key).sign(h)

def dsa_sign(data: str, key) -> bytes:  # DSA or ECC
    h = SHA256.new(data.encode())
    return DSS.new(key, 'fips-186-3').sign(h)

def pretty_print_sig(sig_type: str, sig: bytes) -> None:
    print(f"{sig_type} signature: {sig.hex()}\n")

def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("msg", type=str)
    args = parser.parse_args()

   if not any((path.exists('./rsa.pem'), path.exists('./dsa.pem'), path.exists('./
        rsa_key, dsa_key, ecc_key = generate_keys()
        export_keys(rsa_key, dsa_key, ecc_key)
        print("Generated keys\n")
    else:
        with open("rsa.pem", 'rb') as fp:
            rsa_key = RSA.import_key(fp.read())
        with open("dsa.pem", 'rb') as fp:
            dsa_key = DSA.import_key(fp.read())
        with open("ecc.pem", 'rb') as fp:
```

```
            ecc_key = ECC.import_key(fp.read())
        print("Imported keys\n")

    for key_type, key, sign_func in zip(("RSA","DSA","ECDSA"),
                                        (rsa_key,dsa_key,ecc_key),
                                        (rsa_sign,dsa_sign,dsa_sign)):
        print(f"About to generate signature for {key_type}")
        start_time = time_ns()
        sig = sign_func(args.msg, key)
        delta = time_ns() - start_time
        print(f"Signature took {delta} ns ({delta/1e9} s)")
        pretty_print_sig(key_type, sig)

if __name__ == "__main__":
    main()
```