CSE 4106 : Computer Netowrks Laboratory

Report on Secure Email Communication Simulation:

Man-in-the-Middle Attack Analysis and Certificate-Based Defense

Submitted by

Md Sakibur Rahman

Roll : 2007007

Submitted To:

Md Sakhawat Hossain
Lecturer
Dept. of Computer Science & Engineering

Farhan Sadaf
Lecturer
Dept. of Computer Science & Engineering

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna-9203, Bangladesh

Contents

List of Figures

1 Project Objectives

This project aims to explore and demonstrate the vulnerabilities and defense mechanisms in secure email communication systems through simulation and analysis of Man-in-the-Middle attacks using the OMNeT++ discrete event simulator.

- **SMTP Implementation:** To implement a secure SMTP server for email communication using IMAP and HTTP protocols.
- **Diffie-Hellman Integration:** To integrate Diffie-Hellman key exchange for establishing secure shared keys between client and server.
- **RSA Encryption:** To apply RSA and public key cryptosystems for encrypting and securing email content.
- **Attack Simulation:** To simulate and analyze a man-in-the-middle attack in the context of SMTP communication.
- **Certificate Authentication:** To use digital certificates for authentication and integrity verification to mitigate potential attacks.
- **Performance Assessment:** To assess overall system performance under different attack scenarios and security configurations.

2 Introduction

Context: In the modern digital landscape, secure communication has become paramount as sensitive information is constantly transmitted across networks. Email systems, in particular, handle vast amounts of confidential data daily, making them prime targets for malicious actors.

This project explores the critical vulnerabilities in encrypted communication protocols through the implementation and analysis of a **Man-in-the-Middle (MITM)** attack on an email transmission system. The simulation environment, built using the **OMNeT++ discrete event simulator**, demonstrates how cryptographic protocols, specifically Diffie-Hellman key exchange and RSA encryption, can be compromised when implemented without proper authentication mechanisms.

Attack Scenario

The project illustrates a realistic scenario where a malicious sniffer positioned between a sender (`mta_Client_SS`) and receiver (`mta_Server_RS`) intercepts, decrypts, modifies, and re-encrypts communication without detection by either legitimate party.

Two-Phase Attack Strategy

Phase 1: Key Exchange Interception
During the key exchange phase, the attacker intercepts the Diffie-Hellman handshake messages containing public keys from both parties. By substituting these public keys with its own, the sniffer establishes two separate encrypted sessions—one with the sender and another with the receiver—while both parties believe they are communicating directly with each other.

Phase 2: RSA Exploitation
The attacker attempts to factor the RSA modulus (n) to derive the private key from captured public key parameters. For demonstration purposes, the simulation uses relatively small prime numbers deliberately chosen to be vulnerable to brute-force factorization.

Defense Mechanism

However, the project also implements and demonstrates the effectiveness of countermeasures against such attacks. The most significant defense mechanism explored is the integration of **digital certificates** issued by a trusted Certificate Authority (CA).

Critical Security Insight

When certificates are present in the communication handshake, the MITM attack **fails** because the sniffer cannot forge a valid certificate signature without access to the CA's private key. In this scenario, the sniffer is forced into "transparent mode," where it can only forward messages unmodified, rendering the attack ineffective.

Project Significance

Through detailed logging and real-time visualization of the attack process, this simulation provides valuable insights into both the mechanics of MITM attacks and the importance of authentication in cryptographic protocols. The project serves as an **educational tool** for understanding how:

- Encryption alone is *insufficient* for secure communication
- Authentication and integrity verification through certificates are *crucial* components
- Dual-session architecture enables attackers to decrypt and manipulate messages
- Certificate-based authentication provides robust protection against impersonation

By examining both successful attack scenarios and effective defense mechanisms, this work contributes to a deeper understanding of network security principles and the practical implementation of secure communication systems.

3 Theoretical Background

3.1 Man-in-the-Middle (MITM) Attack

Attack Concept

A **Man-in-the-Middle (MITM) attack** occurs when an attacker (Sniffer) secretly intercepts and controls communication between two parties (Sender and Receiver) who believe they are communicating directly with each other. The attacker positions themselves between the communicating parties, creating two separate encrypted sessions while remaining undetected.

Mathematical Analysis of MITM on Diffie-Hellman

Normal Diffie-Hellman Key Exchange (Without Attack):

Sender and Receiver establish shared secret:

1. Public parameters: generator g and prime p
2. Sender chooses private key a, computes public key: $A = g^a \bmod p$
3. Receiver chooses private key b, computes public key: $B = g^b \bmod p$
4. They exchange public keys: $A \leftrightarrow B$
5. Sender computes: $k_{SR} = B^a \bmod p = (g^b)^a \bmod p = g^{ab} \bmod p$
6. Receiver computes: $k_{SR} = A^b \bmod p = (g^a)^b \bmod p = g^{ab} \bmod p$
7. **Result:** Both share the same secret key $k_{SR} = g^{ab} \bmod p$

MITM Attack on Diffie-Hellman (Sniffer Intercepts):

Sniffer performs two separate DH key exchanges:
Phase 1: Key Substitution

- ▷ Sender computes $A = g^a \bmod p$ and sends to Receiver
- ▷ **Sniffer intercepts**, generates own key: s (private), $S = g^s \bmod p$ (public)
- ▷ **Sniffer substitutes** A with S', sends $S' = g^s \bmod p$ to Receiver
- ▷ Receiver computes $B = g^b \bmod p$ and sends to Sender
- ▷ **Sniffer intercepts**, substitutes B with S'', sends $S'' = g^s \bmod p$ to Sender

Phase 2: Dual Session Establishment

$$\begin{aligned}
\textbf{Sender computes:} \quad & k_{SS} = (S'')^a \bmod p = (g^s)^a \bmod p = g^{sa} \bmod p \\
\textbf{Receiver computes:} \quad & k_{SR} = (S')^b \bmod p = (g^s)^b \bmod p = g^{sb} \bmod p \\
\textbf{Sniffer computes:} \quad & k_{SS} = A^s \bmod p = (g^a)^s \bmod p = g^{as} \bmod p \\
& k_{SR} = B^s \bmod p = (g^b)^s \bmod p = g^{bs} \bmod p
\end{aligned}$$

Phase 3: Message Relay

- ▷ Sender encrypts: $C_1 = E_{k_{SS}}(M)$ using key $g^{as} \bmod p$
- ▷ Sniffer decrypts: $M = D_{k_{SS}}(C_1)$ **(reads plaintext!)**
- ▷ Sniffer re-encrypts: $C_2 = E_{k_{SR}}(M)$ using key $g^{bs} \bmod p$
- ▷ Receiver decrypts: $M = D_{k_{SR}}(C_2)$ successfully

Why the attack works:

- ▷ Sniffer computes $k_{SS} = A^s = (g^a)^s = g^{as} \bmod p$ (shared with Sender)
- ▷ Sniffer computes $k_{SR} = B^s = (g^b)^s = g^{bs} \bmod p$ (shared with Receiver)
- ▷ Sender and Receiver use **different keys**, both known to Sniffer
- ▷ **Critical vulnerability:** No authentication - cannot verify key ownership

Reference: Chapter 19 of Understanding Cryptography by Christof Paar and Jan Pelzl

3.2 Certificate-Based Defense with RSA

RSA Cryptosystem

RSA (Rivest-Shamir-Adleman) is an asymmetric cryptographic algorithm that provides the mathematical foundation for digital signatures and certificates.

RSA Key Generation:

1. Choose two large prime numbers: p and q (e.g., $p = 61$, $q = 53$)
2. Compute modulus: $n = p \times q = 61 \times 53 = 3233$
3. Compute Euler's totient: $\phi(n) = (p - 1)(q - 1) = 60 \times 52 = 3120$
4. Choose public exponent: e (typically 17 or 65537), where $\gcd(e, \phi(n)) = 1$
5. Compute private exponent: $d \equiv e^{-1} \pmod{\phi(n)}$ (e.g., $d = 2753$)
6. **Public key:** $(e, n) = (17, 3233)$
7. **Private key:** $(d, n) = (2753, 3233)$

RSA Encryption/Decryption:

$$\begin{aligned} \text{Encryption:} \quad & C = M^e \bmod n \\ \text{Decryption:} \quad & M = C^d \bmod n \end{aligned}$$

Mathematical Property:

$$(M^e)^d \equiv M^{ed} \equiv M^1 \equiv M \pmod{n}$$

This works because $ed \equiv 1 \pmod{\phi(n)}$ by construction.

How Certificates Prevent MITM Attacks

Certificate-Based Authentication Process:

Step 1: Certificate Issuance

- ▷ Sender requests certificate from CA
- ▷ CA verifies Sender's identity (out-of-band)
- ▷ CA creates certificate with Sender's public key
- ▷ CA signs: $\sigma_S = \text{hash}(\text{Sender\_Cert})^{d_{CA}} \bmod n_{CA}$
- ▷ CA sends $(\text{Sender\_Cert}, \sigma_S)$ to Sender
- ▷ Same process for Receiver: CA sends $(\text{Receiver\_Cert}, \sigma_R)$

Step 2: Authenticated Key Exchange

1. Sender $\rightarrow \{A = g^a \bmod p, \text{Sender\_Cert}, \sigma_S\} \rightarrow$ Receiver
2. Receiver verifies: $\text{hash}(\text{Sender\_Cert}) \stackrel{?}{=} \sigma_S^{e_{CA}} \bmod n_{CA}$
3. If valid: Receiver trusts A belongs to Sender
4. Receiver $\rightarrow \{B = g^b \bmod p, \text{Receiver\_Cert}, \sigma_R\} \rightarrow$ Sender
5. Sender verifies: $\text{hash}(\text{Receiver\_Cert}) \stackrel{?}{=} \sigma_R^{e_{CA}} \bmod n_{CA}$
6. If valid: Sender trusts B belongs to Receiver
7. Both compute: $k_{SR} = g^{ab} \bmod p$ using **authenticated** keys

Step 3: MITM Attack Failure

- ▷ Sniffer intercepts messages containing certificates
- ▷ Sniffer **cannot** create valid certificate for itself (lacks d_{CA})
- ▷ If Sniffer forwards own public key without certificate: rejected
- ▷ If Sniffer tries to modify certificate: signature verification fails
- ▷ Computing d_{CA} from (e_{CA}, n_{CA}) requires factoring n_{CA} (computationally infeasible)
- ▷ **Result:** Attack detected and prevented

Security Foundation

The security of this system relies on two hard mathematical problems:

- **Discrete Logarithm Problem (DLP):** Given g, p, and $g^x \bmod p$, computing x is computationally hard (protects DH private keys)
- **Integer Factorization Problem:** Given $n = pq$, finding p and q is computationally hard (protects RSA private key)

Key Insight: Certificates transform the key exchange problem from *"How do I securely get Bob's public key?"* to *"How do I verify this public key belongs to Bob?"* The latter is solved by the CA's digital signature, which cryptographically binds identity to public key.

4 System Architecture

This OMNeT++ project consists of **15 interconnected modules** that work together to simulate a complete email transmission system with security features. Each module has a specific role in the email delivery pipeline, from composition to final retrieval.

Network Topology

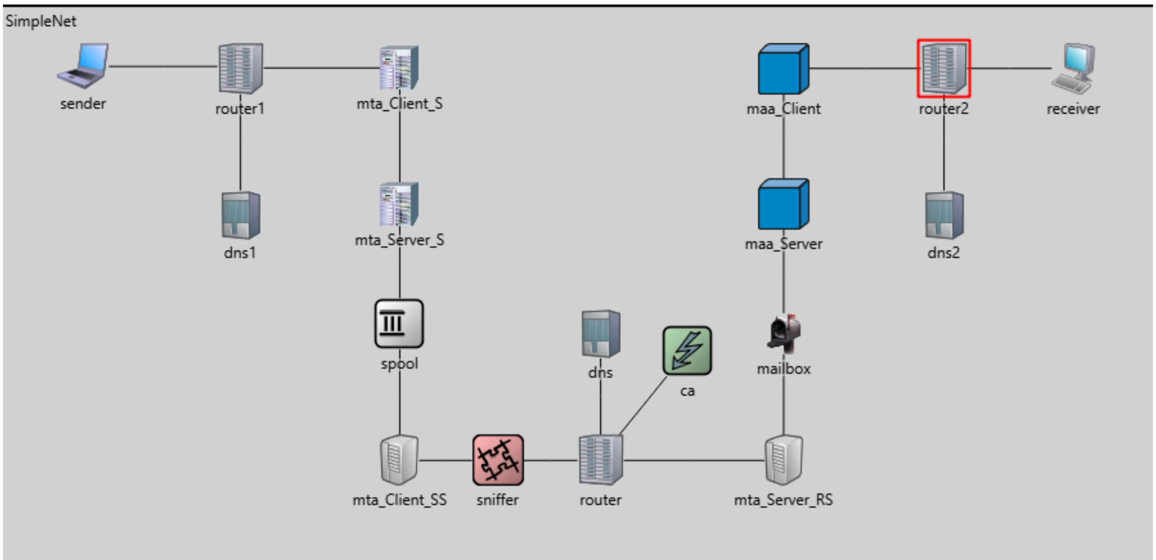

Figure 1: OMNeT++ Network Topology showing all 15 modules including sender, receiver, routers, DNS servers, mail transfer agents (MTA), spool, mailbox, Certificate Authority (CA), and the malicious sniffer positioned between mta_Client_SS and the router.

4.1 Core Communication Modules

1. Sender Module

File: `src/sender.cc` | **Icon:** Laptop device

The Sender module represents an email client that initiates the email transmission process.

Key Functions:

- ▷ Composes email messages with configurable parameters
- ▷ Queries DNS to resolve mail server addresses
- ▷ Submits email via HTTP protocol
- ▷ Initiates the email delivery workflow

2. Receiver Module

File: `src/receiver.cc` | **Icon:** PC device

The Receiver module represents the end user who retrieves and reads emails.

Key Functions:

- ▷ Connects to MAA_Client to retrieve emails
- ▷ Uses IMAP protocol to fetch messages
- ▷ Displays received email content
- ▷ Final destination in the delivery chain

4.2 Infrastructure Services

3. DNS (Domain Name System) Module

File: `src/dns.cc` | **Icon:** Server device

Provides domain name resolution services, mapping logical names to IP addresses.

Key Functions:

- ▷ Receives DNS queries from clients
- ▷ Maintains lookup table for domain-to-IP mapping
- ▷ Returns DNS responses with resolved addresses
- ▷ Supports entire network's name resolution

4. HTTP Module

File: `src/http.cc` | **Icon:** Server device
Provides web-based communication services for email submission and retrieval.
Key Functions:
- Handles HTTP GET requests
- Processes web-based email operations
- Simulates realistic operation delays
- Returns responses with status codes

13. Router Module

File: `src/router.cc` | **Icon:** Router device
Provides packet routing and forwarding services throughout the network.
Key Functions:
- Routes packets based on destination addresses
- Maintains routing tables
- Supports flooding when no route exists
- Enables external attack hooks for testing

4.3 Mail Transfer Agents (MTAs)

5. MTA_Client_S (Sender Side)

File: `src/mta_Client_S.cc` | **Icon:** Old server
First mail transfer agent receiving emails from senders.
Key Functions:
- Receives email submissions via HTTP
- Initiates SMTP communication
- Forwards to MTA_Server_S for relay
- Acts as sender's outgoing mail server

6. MTA_Server_S (Sender Side)

File: `src/mta_Server_S.cc` | **Icon:** Old server
Receives emails from MTA_Client_S and forwards to spool.
Key Functions:
- Accepts incoming SMTP connections
- Validates and processes messages
- Forwards emails to spool for queuing
- Implements SMTP protocol responses

7. Spool Module

File: `src/spool.cc` | **Icon:** Queue block
Provides temporary storage for emails awaiting delivery.
Key Functions:
- ▷ Stores emails temporarily when server unavailable
- ▷ Implements FIFO queue management
- ▷ Forwards to MTA_Client_SS when ready
- ▷ Prevents email loss during network delays

8. MTA_Client_SS (Secondary Sender)

File: `src/mta_Client_SS.cc` | **Icon:** Server device
Intermediate relay client with certificate support.
Key Functions:
- ▷ Retrieves emails from spool
- ▷ Establishes secure Diffie-Hellman connections
- ▷ Supports RSA encryption
- ▷ Can use digital certificates for authentication

Parameters: `useCertificates` (default: false)

9. MTA_Server_RS (Receiver Side)

File: `src/mta_Server_RS.cc` | **Icon:** Server device
Recipient's mail server with encryption support.
Key Functions:
- ▷ Accepts incoming emails
- ▷ Implements Diffie-Hellman key exchange
- ▷ Supports RSA encryption/decryption
- ▷ Validates digital certificates when enabled

Parameters: `maxMessageSizeBytes` (2MB), `useCertificates`

4.4 Mail Storage and Access

10. Mailbox Module

File: `src/mailbox.cc` | **Icon:** Mailbox
Stores received emails and handles IMAP fetch requests.
Key Functions:
- ▷ Stores incoming emails from MTA_Server_RS
- ▷ Sends new mail notifications
- ▷ Responds to IMAP fetch requests
- ▷ Maintains email metadata (sender, subject, flags)

11. MAA_Server (Mail Access Agent Server)

File: `src/maa_Server.cc` | **Icon:** Abstract server
Provides IMAP services for email retrieval.
Key Functions:
- ▷ Receives new mail notifications
- ▷ Handles IMAP requests from clients
- ▷ Fetches emails from mailbox
- ▷ Implements IMAP protocol operations

12. MAA_Client (Mail Access Agent Client)

File: `src/maa_Client.cc` | **Icon:** Abstract server
Client-side mail access agent for the receiver.
Key Functions:
- ▷ Receives new mail notifications
- ▷ Sends IMAP fetch requests
- ▷ Delivers retrieved emails to Receiver
- ▷ Acts as intermediary between receiver and server

4.5 Security Modules

14. MaliciousSniffer Module - ATTACK COMPONENT

File: `src/sniffer.cc` | **Icon:** Red segmented block
Description: Simulates a Man-in-the-Middle attacker positioned between MTA_Client_SS and router.
Attack Capabilities:

- **Key Substitution:** Replaces legitimate public keys with its own
- **Dual Session:** Maintains separate encrypted sessions with both parties
- **Message Modification:** Alters content in transit
- **Cryptographic Attack:** Attempts RSA modulus factorization
- **Traffic Logging:** Records all intercepted communications
- **Transparent Mode:** Becomes passive when certificates detected

15. CA (Certificate Authority) - DEFENSE COMPONENT

File: `src/CA.cc` | **Icon:** Green control block
Description: Issues and signs digital certificates to prevent MITM attacks.
Security Functions:

- Generates RSA key pair for signing certificates
- Issues digital certificates to requesting modules
- Signs certificates with private key
- Provides public key for verification
- Can be enabled/disabled for different scenarios

Parameters: `enabled` (default: false), `certificateValidityPeriod` (3600s)
Security Role: The CA is the *key defense* against MITM attacks. When enabled, it provides cryptographic proof of identity that attackers cannot forge.

4.6 System Workflow

Complete Email Delivery Process

1. **Email Composition:** Sender creates email
2. **Address Resolution:** DNS resolves mail server addresses
3. **Submission:** Sender submits to MTA_Client_S via HTTP
4. **First Relay:** MTA_Client_S forwards to MTA_Server_S via SMTP
5. **Queuing:** MTA_Server_S stores in Spool
6. **Second Relay:** Spool forwards to MTA_Client_SS via Push protocol
7. **Routing:** Router forwards packets (Sniffer may intercept here)
8. **Final Delivery:** MTA_Server_RS delivers to Mailbox
9. **Notification:** Mailbox notifies MAA_Server of new mail
10. **Retrieval:** MAA_Client fetches via IMAP for Receiver

Security Components Summary

Component	Role
MaliciousSniffer	Demonstrates vulnerabilities (Scenario 1)
Certificate Authority	Provides defense (Scenario 2)
Encryption	Diffie-Hellman + RSA protect confidentiality
Authentication	Certificates prevent impersonation

4.7 Scenario 1: MITM Attack Succeeds (No Certificates)

Configuration: `Scenario1_MITM_Attack` in `omnetpp.ini`
Parameters: `**.useCertificates = false` and `**.ca.enabled = false`
Purpose: Demonstrate the vulnerability of unauthenticated key exchange

System Workflow:

Phase 1: Initialization

- `Sender (mta_Client_SS)` at address 300 initializes without certificates
- `Receiver (mta_Server_RS)` at address 500 initializes without certificates
- `Sniffer` positioned between Sender and Router, ready to intercept
- No Certificate Authority involved (disabled in configuration)

Phase 2: Compromised Key Exchange

- `Sender` generates DH parameters (g=5, p=23, private key) and RSA keys (e, n)

- ▷ `Sender` → **DH_HELLO** (DH public=5432, RSA e=17, n=3233) → Router
- ▷ **[SNIFFER INTERCEPTS]** - Message never reaches Receiver
- ▷ **Sniffer** generates own DH parameters and RSA keys
- ▷ **Sniffer** → **Forged DH_HELLO** (Sniffer's keys) → `Receiver`
- ▷ `Receiver` computes shared secret with Sniffer's DH public key (e.g., shared_secret=8)
- ▷ `Receiver` → **DH_HANDSHAKE** (DH public, RSA keys) → Router
- ▷ **[SNIFFER INTERCEPTS]** - Message never reaches Sender
- ▷ **Sniffer** → **Forged DH_HANDSHAKE** (Sniffer's keys) → `Sender`
- ▷ `Sender` computes shared secret with Sniffer's DH public key (e.g., shared_secret=15)

Phase 3: Two Separate Encrypted Sessions Established

- ▷ **Session A:** `Sender` ↔ **Sniffer** (shared secret = 15)
- ▷ **Session B:** **Sniffer** ↔ `Receiver` (shared secret = 8)
- ▷ Both parties believe they have secure end-to-end encryption
- ▷ In reality: Sniffer controls both encryption sessions

Phase 4: SMTP Communication - Email Transmission

- ▷ `Sender` encrypts email using Session A keys (XOR with secret 15)
- ▷ `Sender` → **EHLO, MAIL FROM:** alice@example.com (encrypted) → **Sniffer**
- ▷ **Sniffer** decrypts with Session A key (XOR with secret 15) - **READS CONTENT**
- ▷ **Sniffer** → **RCPT TO:** bob@example.com (encrypted) → **Sniffer**
- ▷ **Sniffer** decrypts and **LOGS: FROM, TO, SUBJECT, CONTENT**
- ▷ **Sniffer** re-encrypts with Session B keys (XOR with secret 8)
- ▷ **Sniffer** → Re-encrypted SMTP messages → `Receiver`
- ▷ `Receiver` successfully decrypts using Session B keys
- ▷ Email delivered normally - **both parties unaware of compromise**

Attack Outcome:

MITM ATTACK SUCCESSFUL

Console Output Shows:

```
INTERCEPTED EMAIL:
  FROM: alice@example.com
  TO: bob@example.com
  SUBJECT: Test Email
  CONTENT: Hello from Sender
```

Impact:

- ▷ Complete confidentiality breach - All email content stolen
- ▷ No authentication - Sniffer impersonates both parties
- ▷ No detection - Neither Sender nor Receiver knows attack occurred
- ▷ Two separate encrypted sessions under attacker's control

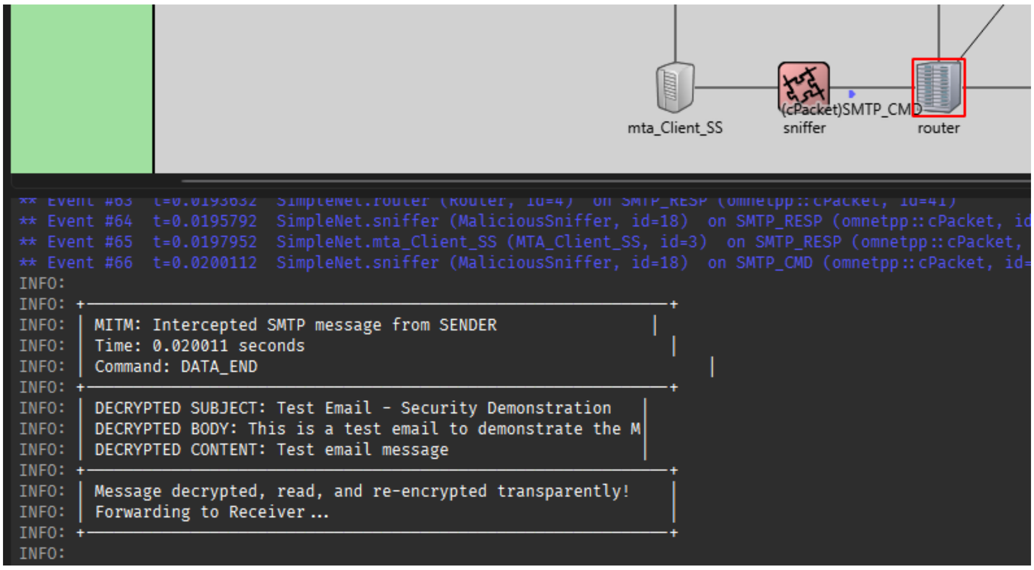

Figure 2: Scenario 1: Sniffer successfully decrypts and reads email content during MITM attack. The console shows intercepted FROM, TO, SUBJECT, and message CONTENT fields.

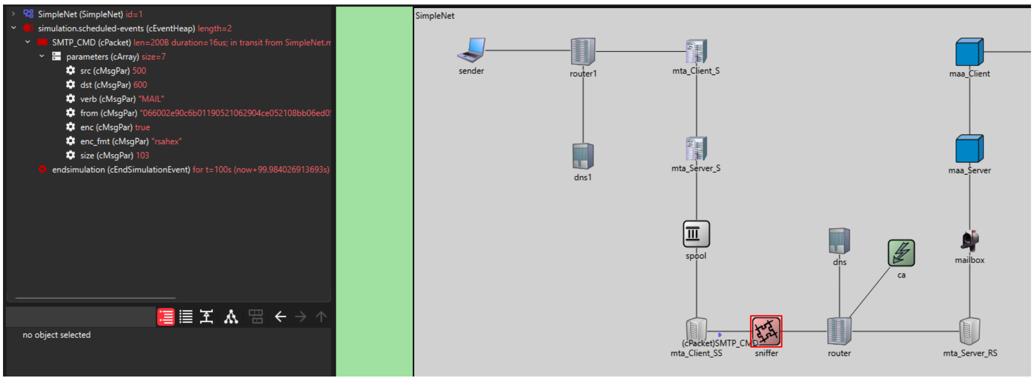

Figure 3: Scenario 1: Email transmission with encryption but without certificate verification. Messages pass through the sniffer who decrypts, reads, and re-encrypts all SMTP communication.

4.8 Scenario 2: Certificates Prevent MITM

Configuration: `Scenario2_With_Certificates` in `omnetpp.ini`
Parameters: `**.useCertificates = true`, `**.ca.enabled = true`, `**.ca.certificateValidityPeriod = 3600s`
Purpose: Demonstrate how PKI and digital certificates prevent MITM attacks

System Workflow:

Phase 1: Certificate Authority Initialization

- ▷ `CA (Certificate Authority)` at address 900 initializes
- ▷ CA generates RSA key pair: Public key (e=17, n=3233), Private key (d=2753)
- ▷ CA announces availability to the network
- ▷ Console displays: `CA INITIALIZED | CA Address: 900 | RSA Public Key: e=17, n=3233`

Phase 2: Certificate Issuance

- ▷ `Sender` → **CERT_REQUEST** (identity, address 300, RSA public, DH public) → `CA`
- ▷ `CA` creates certificate data string: `"Sender|300|17|3233|5432"`
- ▷ `CA` signs certificate using RSA private key (d=2753): `signature = signCertificate(certData`
- ▷ `CA` → **CERT_RESPONSE** (certificate + signature, valid until 3600s) → `Sender`
- ▷ `Sender` stores certificate: `myCertificate = receivedCert`
- ▷ `Receiver` → **CERT_REQUEST** (identity, address 500, RSA public, DH public) → `CA`
- ▷ `CA` issues and signs Receiver's certificate
- ▷ `CA` → **CERT_RESPONSE** (certificate + signature, valid until 3600s) → `Receiver`

- ▷ `Receiver` stores certificate: `myCertificate = receivedCert`

Phase 3: Authenticated Key Exchange with Certificate Validation

- ▷ `Sender` → **DH_HELLO** + **Sender Certificate** + **Signature** → Router
- ▷ **[SNIFFER DETECTS CERTIFICATE]**
- ▷ **Sniffer** attempts to forge certificate but **FAILS** (no CA private key)
- ▷ **Sniffer** console: `Cannot forge valid certificate signature - Attack failed!`
- ▷ **Sniffer** → **Forwards unmodified message** → `Receiver`
- ▷ `Receiver` extracts certificate from DH_HELLO
- ▷ `Receiver` verifies signature: `bool valid = verifyCertificate(cert, signature, CA_e, CA_n)`
- ▷ `Receiver` checks expiration: `if (currentTime < cert.expiryTime) → Valid`
- ▷ `Receiver` console: `Signature valid (CA verification passed)`
- ▷ `Receiver` console: `Certificate not expired`
- ▷ `Receiver` console: `Identity confirmed: Sender`
- ▷ `Receiver` → **DH_HANDSHAKE** + **Receiver Certificate** + **Signature** → Router
- ▷ **Sniffer** → **Forwards unmodified message** → `Sender`
- ▷ `Sender` verifies Receiver's certificate using CA public key
- ▷ `Sender` console: `Signature valid | Certificate not expired | Identity confirmed`
- ▷ Both parties compute shared DH secret using **authentic** public keys

Phase 4: Secure End-to-End Communication

- ▷ `Sender` and `Receiver` establish direct encrypted session (shared secret known only to them)
- ▷ `Sender` → **Encrypted SMTP: EHLO, MAIL, RCPT, DATA** → **Sniffer**
- ▷ **Sniffer cannot decrypt** (lacks shared secret) → forwards encrypted messages
- ▷ **Sniffer** → **Encrypted messages** → `Receiver`
- ▷ `Receiver` decrypts successfully using authentic shared secret
- ▷ Email delivered securely - **MITM attack prevented**

Defense Outcome:

SECURE COMMUNICATION ESTABLISHED!

Console Output Shows:

```
Sniffer: Attempting MITM attack...
    ATTACK FAILED!
    Cannot forge valid certificate
    Receiver rejected fake certificate
    Connection refused

Secure communication established!
```

Protection Achieved:

- ▷ Authentication - Certificates bind public keys to verified identities
- ▷ Integrity - CA signature prevents certificate forgery
- ▷ Confidentiality - End-to-end encryption with authenticated keys
- ▷ Detection - Invalid certificates immediately rejected
- ▷ Trust Model - CA public key known to all parties (out-of-band)

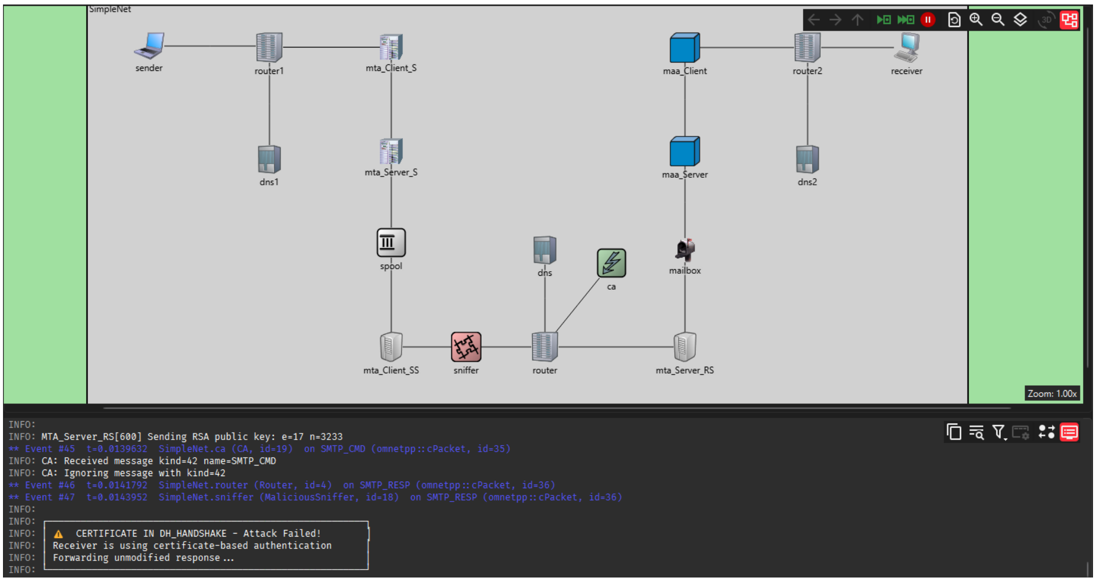

Figure 4: Scenario 2: MITM attack failure. Console output shows the sniffer's attempt to forge certificates failed. The warning message " CERTIFICATE IN DH_HANDSHAKE - Attack Failed!" indicates certificate-based authentication successfully prevented the attack. Receiver uses certificate-based authentication and the sniffer cannot forge valid CA signatures.

Topology reference: Modules and links are defined in 'src/email.ned'. The sniffer is placed between 'mta_Client_SS' and the router; the CA connects to the router and can be enabled/disabled per scenario.

Protocol Overview

This OMNeT++ email delivery project simulates a multi-stage email transmission system using several standard and custom protocols. The system models the complete email lifecycle from composition and sending to delivery and retrieval, implementing six distinct protocols that work together to provide a realistic simulation of internet email infrastructure.

1. DNS (Domain Name System) Protocol

Role:
The DNS protocol resolves domain names (like `user@example.com`) to network addresses (IP addresses) so that the sender can find the recipient's mail server.

How it works in the project:

- When the sender wants to send an email, it first needs to know the address of the recipient's mail server.
- The sender sends a `DNS_QUERY` message to the DNS module.
- The DNS module looks up the domain and replies with a `DNS_RESPONSE` message containing the server's address.
- The DNS module maintains a lookup table or database mapping domain names to IP addresses.

Message Types:

- `DNS_QUERY` - Contains the domain name to be resolved
- `DNS_RESPONSE` - Contains the resolved IP address

Technical Details:

- *Query Processing Time:* Configurable delay to simulate DNS lookup latency
- *Caching:* DNS responses may be cached to improve performance on repeated queries
- *Error Handling:* Returns error codes if domain is not found
- *Module Location:* Implemented in `src/dns.cc` and `src/dns.h`

Real-World Equivalent:
Similar to how email clients query DNS servers to find mail servers (MX records) for recipient domains.

2. HTTP (Hypertext Transfer Protocol)

Role:
Used for submitting emails from the sender to the first mail server (MTA_Client_S), and for communication between mail access agents and clients.

How it works in the project:

- The sender submits the email to the mail server using an `HTTP_GET` message.
- The mail server responds with an `HTTP_RESPONSE` to confirm receipt.
- HTTP may also be used for communication between mail access agents (MAA_Client and MAA_Server).
- Acts as a web-based interface for email submission (similar to webmail services).

Message Types:

- `HTTP_GET` - Request to submit or retrieve email
- `HTTP_RESPONSE` - Confirmation or data response

Technical Details:

- *Connection Type:* Simulates HTTP request-response cycle
- *Payload:* Contains email content or metadata
- *Status Codes:* May include success (200 OK) or error codes (404, 500)
- *Module Location:* Implemented in `src/http.cc` and `src/http.h`
- *Session Management:* May maintain connection state between requests

Real-World Equivalent:
Similar to webmail services (Gmail, Outlook.com) where users submit emails via web browsers using HTTP/HTTPS.

3. SMTP (Simple Mail Transfer Protocol)

Role:
Handles the transfer of emails between mail servers (MTAs). This is the core protocol for email relay across the internet.

How it works in the project:

- After receiving the email, the first mail server (`MTA_Client_S`) uses SMTP to send the email to the next mail server (`MTA_Server_S`).
- The process may involve multiple hops (servers), simulating real-world email relaying.
- SMTP commands and responses are exchanged to ensure the email is delivered correctly.
- Follows a command-response pattern similar to real SMTP (HELO, MAIL FROM, RCPT TO, DATA, QUIT).

Message Types:

- `SMTP_CMD` - Command to send mail (e.g., HELO, MAIL FROM, RCPT TO)
- `SMTP_RESP` - Response to command (e.g., 250 OK, 550 Error)
- `SMTP_SEND` - Actual email data transmission
- `SMTP_ACK` - Acknowledgment of receipt

Technical Details:

- *Command Sequence:*

1. HELO/EHLO - Identify sender server
2. MAIL FROM - Specify sender address
3. RCPT TO - Specify recipient address
4. DATA - Begin message transmission
5. QUIT - Close connection

- *Error Handling:* Supports retry mechanisms for failed deliveries
- *Module Locations:*
  - `src/mta_Client_S.cc` - Sender side MTA
  - `src/mta_Server_S.cc` - Receiver side MTA
  - `src/mta_Client_SS.cc` - Intermediate relay client
  - `src/mta_Server_RS.cc` - Intermediate relay server
- *Port:* Typically uses port 25 (simulated in the project)
- *Authentication:* May include authentication mechanisms (simplified in this simulation)

Real-World Equivalent:
The standard protocol used by all email servers worldwide to exchange emails (RFC 5321).

4. IMAP (Internet Message Access Protocol)

Role:
Allows the receiver to access and retrieve emails from their mailbox. Unlike POP3, IMAP keeps emails on the server.

How it works in the project:

- The receiver's client (`MAA_Client`) sends an `IMAP_FETCH` request to the mailbox via the `MAA_Server`.
- The mailbox responds with an `IMAP_RESPONSE` containing the requested email.
- Supports multiple operations like listing emails, fetching specific messages, marking as read/unread.

Message Types:

- `IMAP_FETCH` - Request to retrieve specific email(s)
- `IMAP_RESPONSE` - Contains the requested email content

Technical Details:

- *Operations Supported:*
  - LIST - List available mailboxes/folders
  - SELECT - Select a specific mailbox
  - FETCH - Retrieve message content
  - SEARCH - Search for messages
  - DELETE - Mark messages for deletion
- *Stateful Protocol:* Maintains connection state between client and server

- *Module Locations:*
  - `src/mailbox.cc` - Stores and manages emails
  - `src/maa_Server.cc` - IMAP server implementation
  - `src/maa_Client.cc` - IMAP client implementation
- *Message Flags:* Supports flags like \Seen, \Answered, \Flagged
- *Synchronization:* Keeps server and client in sync

Real-World Equivalent:
Used by modern email clients (Outlook, Thunderbird, mobile apps) to access emails stored on servers (RFC 3501).

5. Notification Protocol (Custom)

Role:
Notifies the mail access agent and client when new mail arrives in the mailbox. Provides real-time or near-real-time email notifications.

How it works in the project:

- When a new email is stored in the mailbox, the mailbox sends a `NOTIFY_NEWMAIL` message to the `MAA_Server`.
- The MAA_Server may forward this notification to the `MAA_Client` to alert the user.
- Acts as a push notification system to inform users of incoming mail without requiring them to poll the server.

Message Types:

- `NOTIFY_NEWMAIL` - Alert message indicating new email arrival

Technical Details:

- *Trigger:* Automatically sent when mailbox receives new email
- *Content:* May include sender info, subject, timestamp
- *Delivery:* Can be immediate or batched
- *Module Locations:*
  - `src/mailbox.cc` - Generates notifications
  - `src/maa_Server.cc` - Relays notifications
  - `src/maa_Client.cc` - Receives and displays notifications
- *Priority Levels:* May support priority flags for urgent messages
- *User Preferences:* Can be configured to enable/disable notifications

Real-World Equivalent:
Similar to push notifications in mobile email apps, IMAP IDLE command, or Exchange ActiveSync.

6. Push Protocol (Custom)

Role:
Used for pushing emails between certain modules, such as from the mail server to the spool or from the spool to the recipient's mail server. Ensures reliable delivery between internal components.

How it works in the project:

- When an email is ready to be forwarded, a `PUSH_REQUEST` is sent to the next module (e.g., from MTA_Server_S to spool).
- The receiving module replies with a `PUSH_ACK` to confirm receipt.
- Provides a reliable handshake mechanism for internal message passing.
- May include retry logic if acknowledgment is not received.

Message Types:

- `PUSH_REQUEST` - Request to push email to next component
- `PUSH_ACK` - Acknowledgment of successful receipt

Technical Details:

- *Reliability:* Ensures no emails are lost between modules
- *Queue Management:* Works with spool to handle temporary storage
- *Flow Control:* Prevents overwhelming downstream modules
- *Module Locations:*
  - `src/mta_Client_S.cc` - Initiates push
  - `src/mta_Server_S.cc` - Receives and forwards
  - `src/spool.cc` - Intermediate storage
  - `src/mta_Client_SS.cc` - Relay forwarding
- *Timeout Handling:* Retries if ACK not received within timeout
- *Ordering:* Maintains FIFO (First In, First Out) order
- *Buffer Management:* Manages internal buffers to prevent overflow

Real-World Equivalent:
Similar to internal queuing systems in mail servers (like Postfix queue or Sendmail queue), ensuring reliable message transfer between components.

Protocol Interaction Flow

The following describes the complete message flow through the system:

1. **Address Resolution:** Sender queries DNS for recipient's mail server address
2. **Email Submission:** Sender submits email via HTTP to MTA_Client_S
3. **Mail Relay:** Email is relayed through multiple MTAs using SMTP
4. **Spooling:** Email is temporarily stored in spool using Push protocol
5. **Final Delivery:** Email is delivered to recipient's mailbox

6. **Notification:** Mailbox notifies MAA_Server and MAA_Client of new mail
7. **Retrieval:** Receiver fetches email using IMAP protocol

Discussion

This project explores the strengths and weaknesses of cryptographic protocols in real-world networks through a simulated Man-in-the-Middle (MITM) attack on an email system. Using OMNeT++, we demonstrated how attackers can exploit the lack of authentication in protocols like Diffie-Hellman and RSA to intercept and manipulate sensitive data. The simulation revealed attack techniques such as key substitution and brute-force factorization. It also highlighted the importance of authentication, as encryption alone is insufficient to secure communication. The project emphasized the role of certificate-based authentication, showing that digital certificates prevent MITM attacks by ensuring secure communication between parties. This finding reinforces the need for a multi-layered security approach combining encryption, authentication, and integrity mechanisms.

Conclusion

This project demonstrates that the security of digital communication systems depends not only on strong cryptographic algorithms but also on robust authentication mechanisms. By simulating a MITM attack, we showed that encryption without authentication can be easily compromised. The successful mitigation of the attack through certificate-based authentication stresses the importance of using trusted authorities and digital certificates to prevent impersonation. The results underscore the necessity of incorporating both encryption and authentication for secure systems, and the need for careful protocol design to ensure resilience against evolving security threats.

5 References

References

[1] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–59.
Available: https://omnetpp.org/

[2] "OMNeT++ Discrete Event Simulator - User Manual, Version 6.2," OMNeT++ Documentation, 2025.
Available: https://doc.omnetpp.org/omnetpp/manual/

[3] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321, Internet Engineering Task Force, Oct. 2008.
Available: https://www.rfc-editor.org/rfc/rfc5321

[4] W. Diffie and M. Hellman, "New directions in cryptography," in *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644-654, November 1976.
DOI: 10.1109/TIT.1976.1055638

[5] "SMTP Protocol - Network Security Presentation," Google Slides Presentation.
Available: https://docs.google.com/presentation/d/120bFVOPYJSHXJ41zwws-RYSdqnI8ezrY

[6] "MITM Attack and Certificate-Based Defense," Google Slides Presentation.
Available: https://docs.google.com/presentation/d/1UAqh5GC-xQAV8R46ZLcln4d_lT-Wf0OC