NETSEC TP3: Practical Uses of Public and Private Keys Beyond TLS

Report for Exercises 1 & 2 & 3

Assem Chebly

November 26, 2025

Executive Summary

This report presents three essential exercises demonstrating Public Key Infrastructure (PKI) applications: SSH key-based authentication (Exercise 1), mutual TLS (mTLS) client-server communication (Exercise 2), and OpenVPN-based VPN setup with PKI authentication (Exercise 3). Each exercise highlights the practical use of asymmetric cryptography to secure different network layers.

Exercise 1 replaces passwords with RSA key pairs for secure SSH access. Exercise 2 uses X.509 certificates to mutually authenticate clients and servers in a secure API context. Exercise 3 establishes encrypted network tunnels with certificate-based user access and firewall-enforced controls.

Together, they illustrate PKI's role in providing authentication, confidentiality, integrity, and fine-grained authorization—from individual remote logins through secure APIs to encrypted site-to-site networking—showing PKI's flexibility in modern network security solutions.

1 Exercise 1: SSH Key-Based Access

1.1 Objective

The goal is to configure key-based authentication to a remote SSH server, replacing password-based authentication with public key cryptography.

1.2 Implementation Steps

Key generation. An RSA key pair was generated on the client:

```
ssh-keygen -t rsa -b 4096 -f ~/.ssh/id_rsa
```

This produced:

- `~/.ssh/id_rsa`: private key kept secret on the local machine.
- `~/.ssh/id_rsa.pub`: public key later deployed to the server.

Server setup. The lab environment started an SSH server in a Docker container listening on port 2222. A container identifier was stored in `cont_id.txt` for reference.

Key deployment. The client public key was copied to the server:

```
ssh-copy-id -p 2222 student@0.0.0.0
```

This added the public key to the server's `~/.ssh/authorized_keys` for the `student` account.

Authentication test. Passwordless login was verified by connecting:

```
ssh student@0.0.0.0 -p 2222
```

The connection succeeded without prompting for a password, confirming key-based authentication.

1.3 Answers to Exercise Questions

1. Does the server know the identity of the client?

Yes. The server associates a client identity with a specific public key stored in `authorized_keys`. During login, the server sends a challenge which the client signs with its private key. The server verifies the signature using the stored public key. If verification succeeds, the server is assured that the client controls the private key corresponding to that public key, and therefore treats the client as the legitimate user linked to that key.

2. How does the client know that the server is legitimate?

The client verifies the SSH server using the server's host key. On the first connection the client is shown the host key fingerprint and must decide whether to trust it (Trust On First Use). The fingerprint is then saved to `~/.ssh/known_hosts`. On subsequent connections, the client automatically checks that the presented host key matches the stored one. If the key changes unexpectedly, SSH warns about a possible man-in-the-middle attack.

3. What are the possible vulnerabilities of this authentication mechanism?

Important vulnerabilities and mitigations include:

- **Private key compromise:** if an attacker obtains the private key, they can authenticate as the user. Keys should be stored securely, protected by file permissions and preferably a passphrase.
- **Weak keys or bad randomness:** using short keys or weak randomness reduces security. Strong keys (e.g. 4096-bit RSA or Ed25519) generated by a trusted implementation should be used.
- **Unprotected private key files:** if private key files are world-readable, other local users can steal them. The `~/.ssh` directory and key files should have restrictive permissions (e.g. 700 for the directory, 600 for the files).
- **First-connection trust (TOFU):** if the fingerprint is not verified out-of-band on first use, a man-in-the-middle could present a fake key. Ideally, the host key fingerprint is checked via a secure channel before trusting it.
- **Server-side misconfiguration:** if `authorized_keys` is editable by other users, they could add their own keys. Correct ownership and permissions on user accounts and SSH configuration are required.

2 Exercise 2: Mutual TLS (mTLS) Clients and Server

2.1 Objective

The objective is to implement a secure payment system using mutual TLS (mTLS), where both client and server authenticate each other using X.509 certificates. The server only processes payment requests from authorized clients and enforces account balance constraints.

2.2 System Overview

- **BankServer**: central payment processor, implemented as an HTTPS server with mTLS.
- **Alice**: client with an initial balance of 1000.
- **Bob**: client with an initial balance of 150.
- **Certificate Authority (CA)**: issues and signs server and client certificates.

The server offers a `/payment?amount=<number>` endpoint that:

- Authenticates the client via its TLS certificate.
- Maps the client identity from the certificate's Common Name (CN) to an account.
- Checks authorization and available balance.
- Deducts the amount when valid and returns a response.

2.3 PKI Setup

All keys and certificates were created in a `certs/` directory.

Certificate Authority

A root CA key and certificate were created:

```
openssl genrsa -out ca-key.pem 4096
openssl req -x509 -new -key ca-key.pem -sha256 -days 730 \
  -out ca-cert.pem \
  -subj "/C=FR/ST=Paris/L=Paris/O=BankCompany/CN=MyRootCA"
```

Server Certificate with SAN for localhost

A configuration file `server_openssl.cnf` ensures that the certificate has CN = `localhost` and a Subject Alternative Name (SAN) for hostname verification:

```
[req]
default_bits       = 4096
prompt             = no
default_md         = sha256
req_extensions     = req_ext
distinguished_name = dn

[dn]
C=FR
ST=Paris
L=Paris
O=BankCompany
CN=localhost

[req_ext]
subjectAltName = @alt_names

[alt_names]
DNS.1 = localhost
```

The server key, CSR and certificate:

```
openssl genrsa -out server-key.pem 4096
openssl req -new -key server-key.pem -out server.csr \
  -config server_openssl.cnf
openssl x509 -req -in server.csr -CA ca-cert.pem -CAkey ca-key.pem \
  -CAcreateserial -out server-cert.pem -days 365 -sha256 \
  -extfile server_openssl.cnf -extensions req_ext
```

Client Certificates (Alice and Bob)

For Alice:

```
openssl genrsa -out alice-key.pem 4096
openssl req -new -key alice-key.pem -out alice.csr \
  -subj "/C=FR/ST=Paris/L=Paris/O=BankCompany/CN=alice"
openssl x509 -req -in alice.csr -CA ca-cert.pem -CAkey ca-key.pem \
  -CAcreateserial -out alice-cert.pem -days 365 -sha256
```

For Bob (with CN = `bob`):

```
openssl genrsa -out bob-key.pem 4096
openssl req -new -key bob-key.pem -out bob.csr \
  -subj "/C=FR/ST=Paris/L=Paris/O=BankCompany/CN=bob"
openssl x509 -req -in bob.csr -CA ca-cert.pem -CAkey ca-key.pem \
  -CAcreateserial -out bob-cert.pem -days 365 -sha256
```

2.4 Server Implementation

The server is implemented in Python using `HTTPServer` and `BaseHTTPRequestHandler` with an SSL context configured for mutual TLS.

Core Data Structures

```
BALANCES = {
    "alice": 1000,
    "bob": 150
}
```

mTLS Configuration

The TLS context is configured to:

- Present the server certificate and private key.
- Trust the CA for client certificates.
- Require a valid client certificate.

```
context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="./certs/server-cert.pem",
                        keyfile="./certs/server-key.pem")
context.load_verify_locations(cafile="./certs/ca-cert.pem")
context.verify_mode = ssl.CERT_REQUIRED
httpd.socket = context.wrap_socket(httpd.socket, server_side=True)
```

/payment Endpoint Logic

The `/payment` endpoint:

1. Extracts the `amount` query parameter and parses it as a positive integer.
2. Uses `self.connection.getpeercert()` to read the client's certificate.
3. Extracts the Common Name (CN) from the certificate subject as the username.
4. Verifies that the user is in `BALANCES`.
5. Checks that the requested amount does not exceed the user's balance.
6. Deducts the amount and returns a success message with the new balance, or returns an error.

2.5 Client Implementations

The clients use the Python `requests` library with mutual TLS parameters.

Alice Client

```
response = requests.post(
    "https://localhost:8080/payment?amount=500",
    cert=("./certs/alice-cert.pem", "./certs/alice-key.pem"),
    verify="./certs/ca-cert.pem"
)
```

Bob's client is analogous, but uses Bob's certificate and key.

2.6 Testing and Results

Alice

- Payment of 500: accepted, new balance 500.
- Subsequent payment of 600: rejected due to insufficient funds.

Bob

- Payment of 100: accepted, new balance 50.
- Second payment of 100: rejected due to insufficient funds.

2.7 Security Analysis

The mTLS system provides:

- **Mutual authentication:** server and clients both present certificates signed by the CA.
- **Confidentiality:** TLS encrypts all traffic between clients and server.
- **Integrity:** TLS prevents modification of messages in transit.
- **Authorization:** only clients whose CN matches a known account in `BALANCES` can perform payments.
- **Replay resistance:** TLS sessions and application logic together reduce the risk of replayed payment requests.

Potential risks include:

- Private key compromise on client or server.
- Insecure storage of CA key.
- Misconfiguration of certificate validation or CA trust store.

3 Exercise 3: Set Up a VPN Server

4 Objective

The objective of this exercise was to implement an OpenVPN server on the Cloud Gateway (`gwcloudA`) to bridge the Site A and Site B networks with the Cloud network (`10.12.0.0/16`).

The security requirements were:

- **Developer A (Site A):** Must have access to both **Development** (`10.12.0.20`) and **Production** (`10.12.0.30`) servers.
- **Developer B (Site B):** Must have access **only** to the **Development** server. Access to **Production** must be blocked.

5 Implementation Steps & Commands

5.1 Public Key Infrastructure (PKI) Setup

We established a trusted PKI using `openssl` directly on `gwcloudA`.

Commands used to generate the keys:

```
mkdir -p /exports/keys /exports/configs /exports/evidence

openssl req -new -x509 -days 365 -nodes -out /exports/keys/ca.crt -keyout /expor

openssl dhparam -out /exports/keys/dh.pem 2048

openssl req -new -newkey rsa:2048 -nodes -keyout /exports/keys/server.key -out /
openssl x509 -req -in /exports/keys/server.csr -CA /exports/keys/ca.crt -CAkey /

openssl req -new -newkey rsa:2048 -nodes -keyout /exports/keys/developerA.key -o
openssl x509 -req -in /exports/keys/developerA.csr -CA /exports/keys/ca.crt -CAk

openssl req -new -newkey rsa:2048 -nodes -keyout /exports/keys/developerB.key -o
openssl x509 -req -in /exports/keys/developerB.csr -CA /exports/keys/ca.crt -CAk
```

5.2 Server Configuration

Configuration File (`/exports/configs/server.conf`):

```
port 1194
proto udp
dev tun
ca /exports/keys/ca.crt
cert /exports/keys/server.crt
key /exports/keys/server.key
dh /exports/keys/dh.pem
server 10.8.0.0 255.255.255.0
topology subnet
push "route 10.12.0.0 255.255.0.0"
client-config-dir /etc/openvpn/ccd
keepalive 10 120
cipher AES-256-CBC
user nobody
```

```
group nobody
persist-key
persist-tun
verb 3
```

Commands to set up Static IPs:

```
mkdir -p /etc/openvpn/ccd
echo "ifconfig-push␣10.8.0.10␣255.255.255.0" > /etc/openvpn/ccd/developerA
echo "ifconfig-push␣10.8.0.20␣255.255.255.0" > /etc/openvpn/ccd/developerB
```

Command to start the Server:

```
openvpn --config /exports/configs/server.conf --daemon
```

5.3 Firewall & Access Control (`gwcloudA`)

We utilized `iptables` to route traffic and enforce the specific access restriction for Developer B.

Commands used:

```
sysctl -w net.ipv4.ip_forward=1
iptables -A INPUT -i tun0 -j ACCEPT
iptables -A FORWARD -i tun0 -j ACCEPT
iptables -A FORWARD -i eth1 -o tun0 -j ACCEPT
iptables -t nat -A POSTROUTING -s 10.8.0.0/24 -o eth1 -j MASQUERADE
iptables -I FORWARD -s 10.8.0.20 -d 10.12.0.30 -j DROP
```

5.4 Client Configuration

Below are the full configuration files used for both clients.

Developer A Configuration (`/exports/configs/developerA.ovpn`):

```
client
dev tun
proto udp
remote 100.66.0.3 1194
resolv-retry infinite
nobind
persist-key
persist-tun
cipher AES-256-CBC
ca /exports/keys/ca.crt
cert /exports/keys/developerA.crt
key /exports/keys/developerA.key
```

Developer B Configuration (`/exports/configs/developerB.ovpn`):

```
client
dev tun
proto udp
remote 100.66.0.3 1194
resolv-retry infinite
nobind
persist-key
persist-tun
cipher AES-256-CBC
```

```
ca /exports/keys/ca.crt
cert /exports/keys/developerB.crt
key /exports/keys/developerB.key
```

Command to connect (on Client containers):

```
mkdir -p /dev/net && mknod /dev/net/tun c 10 200 && chmod 600 /dev/net/tun
openvpn --config /exports/configs/developerA.ovpn --daemon
```

(Replace `developerA.ovpn` with `developerB.ovpn` for the second client)

6 Verification & Evidence

6.1 Connectivity Testing (Ping Analysis)

We verified the configuration by running ping tests from both developer containers.

- **Developer A (Site A):**
  - **Result:** `SUCCESS` for both Production (`10.12.0.30`) and Development (`10.12.0.20`) servers.
  - **Evidence:** `ping_test_devA.png`.
- **Developer B (Site B):**
  - **Result:** `SUCCESS` for Development (`10.12.0.20`).
  - **Result:** `FAILURE` (Blocked) for Production (`10.12.0.30`). The ping resulted in 100% packet loss, confirming the firewall rule is active.
  - **Evidence:** `ping_test_devB.png`.

7 Verification & Evidence

7.1 Connectivity Testing (Ping Analysis)

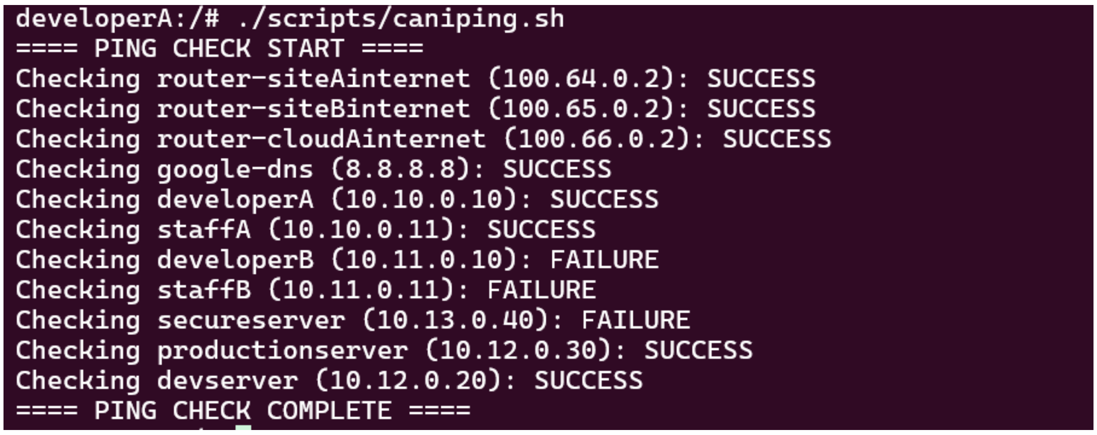
We verified the configuration by running ping tests from both developer containers.

Developer A (Site A)

- **Result:** `SUCCESS` for both Production (`10.12.0.30`) and Development (`10.12.0.20`) servers.

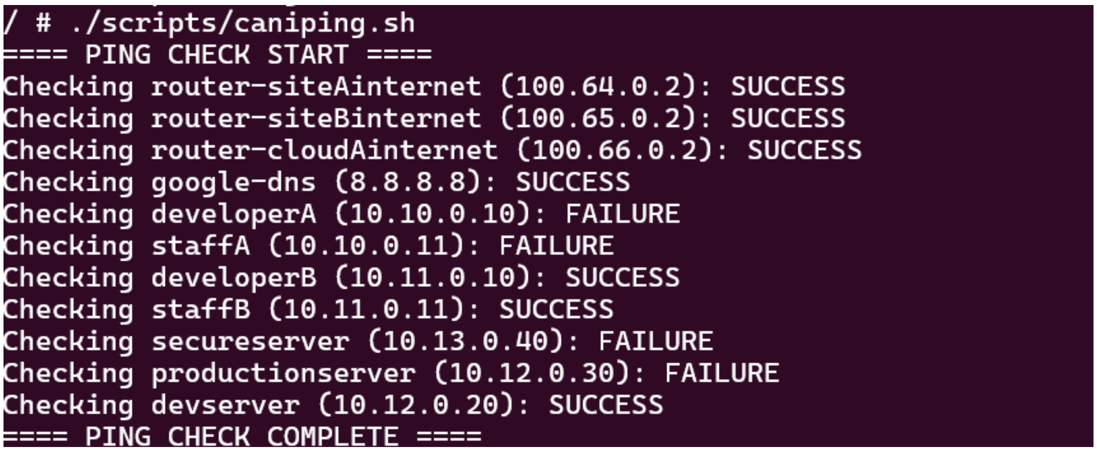
Developer B (Site B)

- **Result:** `SUCCESS` for Development (`10.12.0.20`).
- **Result:** `FAILURE` (Blocked) for Production (`10.12.0.30`). The ping resulted in 100% packet loss, confirming the firewall rule is active.



```
developerA:/# ./scripts/caniping.sh
==== PING CHECK START ====
Checking router-siteAinternet (100.64.0.2): SUCCESS
Checking router-siteBinternet (100.65.0.2): SUCCESS
Checking router-cloudAinternet (100.66.0.2): SUCCESS
Checking google-dns (8.8.8.8): SUCCESS
Checking developerA (10.10.0.10): SUCCESS
Checking staffA (10.10.0.11): SUCCESS
Checking developerB (10.11.0.10): FAILURE
Checking staffB (10.11.0.11): FAILURE
Checking secureserver (10.13.0.40): FAILURE
Checking productionserver (10.12.0.30): SUCCESS
Checking devserver (10.12.0.20): SUCCESS
==== PING CHECK COMPLETE ====
```

Figure 1: Verification of connectivity for Developer A (Access to both servers)



```
/ # ./scripts/caniping.sh
==== PING CHECK START ====
Checking router-siteAinternet (100.64.0.2): SUCCESS
Checking router-siteBinternet (100.65.0.2): SUCCESS
Checking router-cloudAinternet (100.66.0.2): SUCCESS
Checking google-dns (8.8.8.8): SUCCESS
Checking developerA (10.10.0.10): FAILURE
Checking staffA (10.10.0.11): FAILURE
Checking developerB (10.11.0.10): SUCCESS
Checking staffB (10.11.0.11): SUCCESS
Checking secureserver (10.13.0.40): FAILURE
Checking productionserver (10.12.0.30): FAILURE
Checking devserver (10.12.0.20): SUCCESS
==== PING CHECK COMPLETE ====
```

Figure 2: Verification of restrictions for Developer B (Blocked from Production)

8 Submitted Files

The following files are included in the `exercise-3` folder of the submission:

- **`keys/`**: Directory containing the generated CA, Server, and Client certificates/keys.
- **`configs/`**: `server.conf`, `developerA.ovpn`, and `developerB.ovpn`.
- **`evidence/`**:
  - `gwcloudA_iptables.rules`: Export of the active firewall rules.
  - `vpn_traffic.pcap`: Network capture proving encryption.

Conclusion

Exercises 1, 2, and 3 collectively demonstrate how **public and private keys** (asymmetric cryptography) are used to build secure, scalable, and robust solutions across different layers of network security:

- **SSH key-based authentication** eliminates weaknesses of passwords for remote access, protecting against brute force and theft, provided keys and known_hosts are managed securely.

- **Mutual TLS (mTLS)** uses certificates to build trust between clients and servers, enabling fine-grained authorization and confidentiality for APIs such as payment systems.
- **VPN with PKI authentication** extends trust to network connectivity, creating encrypted tunnels that link distributed infrastructure, with authorization possible at both the certificate and firewall level.

Each approach highlights that secure key and certificate management, alongside correct deployment and access policies, is essential. Public key systems not only authenticate, but also provide confidentiality, integrity, and flexible authorization—foundations for defending modern systems against evolving threats. As organizations grow, scalable PKI deployments (as seen in Exercises 2 and 3) become critical to handle complex trust relationships, enforce security policies, and enable safe collaboration across users and sites.