Activity V : Public Key Infrastructure

Overview

In this activity, you will learn the fundamentals of Public Key Infrastructure. We will need the following tools:

A. OpenSSL. On Linux and Mac OS X, the OpenSSL is installed by default. For Windows, you may download it from https://wiki.openssl.org/index.php/Binaries

B. Python with PyOpenSSL and pem to do our exercise. If you python does not come with PyOpenSSL and pem, you may install it with pip

```
$ pip install pyopenssl
$ pip install pem
```

C. You also need ca-certificates.crt from your OS (e.g. /etc/cacerts/ca-certificates.crt in Linux) or take it from the course web.

Exercise

Issuing the following command. `openssl s_client -connect twitter.com:443`
Once connected, you may try
GET / HTTP/1.0
[Enter twice]
(Note that the server may return HTTP 404. This is completely normal since we did not send a request for a valid resource.)
Repeat the same step again, now with `openssl s_client -connect twitter.com:443 -CAfile ca-certificates.crt`
This command basically connects to port 443 (HTTPS) with the TLS/SSL. This is like a standard telnet command, but with openssl performing the encryption for you.

Q1.

From the two given openssl commands, what is the difference?
Note: If your operating system does not show any error in the first command, try `openssl s_client -connect twitter.com:443 -CApath empty_dir`. If the results are still the same, your system is not reliable. You may ignore this exercise. (Modern versions of Mac OS X will always read CA from keychains. There is no intuitive way to turn it off.)

Answer:

- `openssl s_client -connect twitter.com:443` -> The handshake succeeds and verification shows **OK** on your system. That's because OpenSSL is implicitly using your OS trust store (Mac/Linux often auto-load roots from Keychain or `/etc/ssl/certs`).

- `openssl s_client -connect twitter.com:443 -CAfile ca-certificates.crt` -> The handshake also succeeds and verification is explicitly **OK**. Here you've told OpenSSL exactly which trust anchor bundle to use.

- `openssl s_client -connect twitter.com:443 -CApath empty_dir` --> The handshake still works cryptographically, but verification fails with `Verify return code: 20 (unable to get local issuer certificate)`. This shows that without a trusted CA file/path, OpenSSL cannot complete the certificate chain.

The difference is that the **first command** relies on the default trust sources of your OS/OpenSSL build, while the **second command** explicitly provides a CA bundle (ca-certificates.crt) for chain validation. The practical effect is in certificate verification:

- Without a trusted CA path/file, OpenSSL cannot validate the chain and reports a **verify error**.

- With the CA bundle, OpenSSL can validate the chain from twitter.com → Let's Encrypt intermediate → ISRG Root X1, and reports **OK**.

This demonstrates how PKI depends on having the correct **trust anchors (root CAs)** available

Q2.

What does the error (verify error) in the first command mean? Please explain.

Answer:
```
Verification error: unable to get local issuer certificate
Verify return code: 20 (unable to get local issuer certificate)
```
That error means OpenSSL could not build a complete chain of trust from the server's certificate (twitter.com → Let's Encrypt "E6") to a trusted **root CA**. The server did send its own certificate and its intermediate, but because your client was given an **empty trust store**, there was no known root CA to validate against.

In PKI terms:

- **Leaf certificate** = proves the site's identity.

- **Intermediate certificate** = signed by a higher authority, bridges the leaf to the root.

- **Root CA certificate** = ultimate trust anchor, must already be trusted on the client.

Without that root in the local trust store, OpenSSL cannot verify the authenticity of the chain, so it reports a verification error.

Q3.

Copy the server certificate (beginning with -----BEGIN CERTIFICATE----- and ending with -----END CERTIFICATE-----) and store it as twitter_com.cert. Use the command openssl x509 -in twitter_com.cert -text to show a text representation of the certificate content. Briefly explain what is stored in an X.509 certificate (i.e. data in each field).

Answer:

Version: Version 3 — current standard.

Serial Number: Unique identifier for this certificate.

Signature Algorithm: ecdsa-with-SHA384 — algorithm used by the issuer to sign this certificate.

Issuer: C=US, O=Let's Encrypt, CN=E6 — the CA that issued and signed the cert.

Validity:

- Not Before: Aug 19 2025
- Not After: Nov 17 2025
  Defines the time window when the cert is valid.

Subject: CN=twitter.com — the entity this cert belongs to.

Subject Public Key Info: Public key algorithm (EC P-256) and the actual public key.

X.509 Extensions:

- **Key Usage**: Digital Signature (what the key can do).
- **Extended Key Usage**: TLS Web Server Authentication, TLS Web Client Authentication.
- **Basic Constraints**: CA:FALSE → not a CA, just an end-entity cert.
- **Subject Key Identifier** & **Authority Key Identifier**: Key fingerprints used to build the chain.
- **Authority Information Access (AIA)**: Link to download the issuer's (intermediate) cert.
- **Subject Alternative Name (SAN)**: Lists all DNS names covered (e.g., twitter.com, *.twitter.com).
- **Certificate Policies**: OID referencing policy under which it was issued.
- **CRL Distribution Points**: Where revocation lists can be retrieved.

- **Signed Certificate Timestamps (SCTs)**: Evidence of Certificate Transparency logs.

Signature: Digital signature from the issuer, proving authenticity.

Q4.

From the information in exercise 3, is there an intermediate certificate? If yes, what purpose does it serve? Hint: Look for an issuer and download the intermediate certificate. You may use the command openssl x509 -inform der -in intermediate.cert -text to show the details of the intermediate certificate. (Note that the -inform der is for reading the DER file. The default file format for x509 is the PEM file.)

Answer:

```
subject=CN=twitter.com
issuer=C=US, O=Let's Encrypt, CN=E6
```

Yes, there is an intermediate certificate. Twitter's cert is issued by "Let's Encrypt E6," which is signed by the ISRG Root X1 root CA. The intermediate links the website's certificate to the trusted root, allowing the root key to remain offline for security while still enabling trust verification.

Q5.

Is there an intermediate CA, i.e. is there more than one organization involved in the certification? Say why you think so.

Answer:

Yes. More than one organization is involved: Twitter (the subject), Let's Encrypt (the intermediate CA), and ISRG (the root CA). The presence of Let's Encrypt E6 as issuer of Twitter's cert shows that an intermediate CA sits between the website and the root.

Q6.

What is the role of ca-certificates.crt?

Answer:

The file ca-certificates.crt stores trusted root CA certificates. It serves as the local trust store that OpenSSL uses to verify certificate chains, ensuring that a server's certificate can be linked to a trusted root.

Q7.

Explore the ca-certificates.crt. How many certificates are in there? Give the command/method you have used to count.

Answer:

```
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 % grep -c "BEGIN CERTIFICATE"
ca-certificates.crt
127
```

There are 127 certificates in ca-certificates.crt. I counted them using:
```
grep -c "BEGIN CERTIFICATE" ca-certificates.crt
```

Q8.

Extract a root certificate from ca-certificates.crt. Use the openssl command to explore the details. Do you see any Issuer information? Please compare it to the details of twitter's certificate and the details of the intermediate certificate.

Answer:
```
# Split the bundle into individual PEM files: root_001.pem, root_002.pem, ...
awk 'BEGIN{n=0;f=0} /BEGIN CERTIFICATE/{f=1; n++;
file=sprintf("root_%03d.pem",n)} f{print > file} /END CERTIFICATE/{f=0}' ca-
certificates.crt

# Inspect one (e.g., the first)
openssl x509 -in root_001.pem -text -noout
```

Twitter leaf cert
Subject = CN=twitter.com
Issuer = CN=E6, O=Let's Encrypt, C=US
—> Signed by an **intermediate**.

Intermediate (Let's Encrypt E6)
Subject = CN=E6, O=Let's Encrypt, C=US
Issuer = CN=ISRG Root X1, O=Internet Security Research Group, C=US
→ Signed by the **root ISRG**.

Root (ACCVRAIZ1)
Subject = Issuer = ACCVRAIZ1 (self-signed).

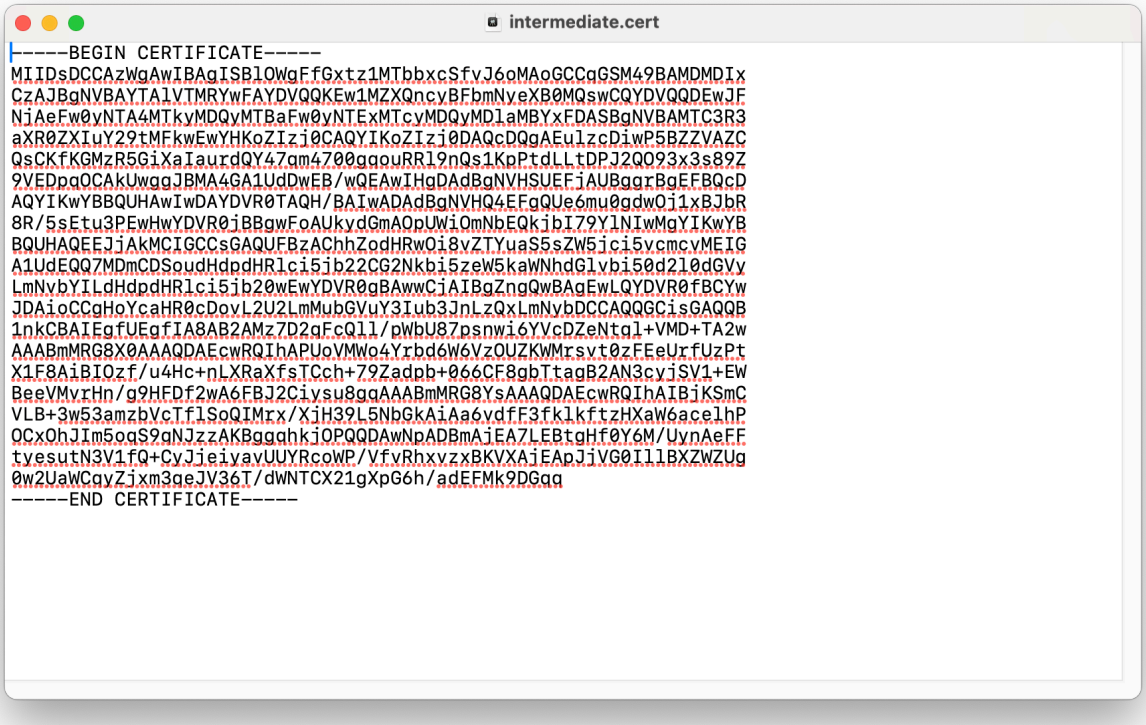
→ Trust anchor in the system store.

When inspecting a root certificate from ca-certificates.crt, the Issuer and Subject are identical, meaning it is self-signed. In contrast, Twitter's certificate shows Subject = twitter.com and Issuer = Let's Encrypt E6, while the intermediate E6 certificate's Issuer is ISRG Root X1. This demonstrates the chain of trust: leaf → intermediate → root, with only the root being self-issued.

Q9.

If the intermediate certificate is not in a PEM format (text readable), use the command to convert a DER file (.crt .cer .der) to PEM file. `openssl x509 -inform der -in certificate.cer -out certificate.pem`. (You need the pem file for exercise 10.)

Answer:



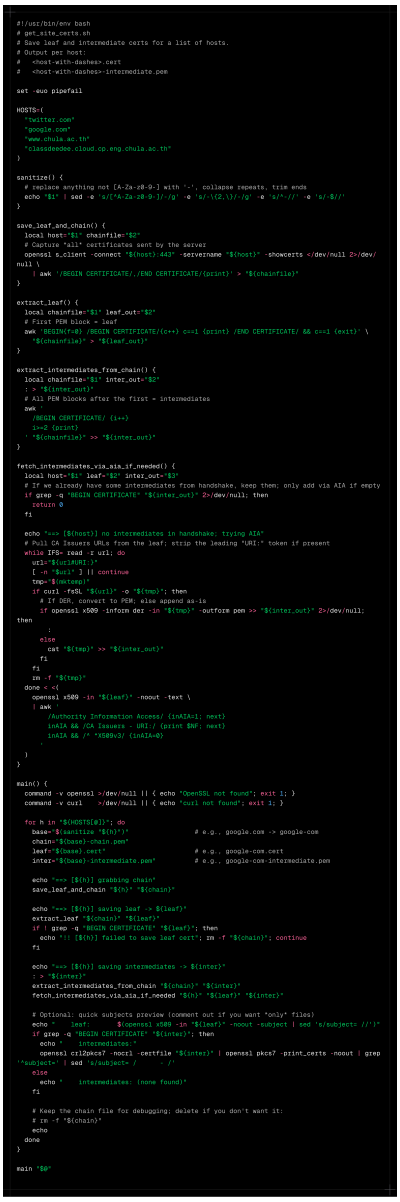
Q10.

From the given python code, implement the certificate validation. Use your program to verify the certificates of: Twitter, google, www.chula.ac.th, classdeedee.cloud.cp.eng.chula.ac.th

Answer:

```
.../repositories/computer-security/activity-05

google-com.cert: OK
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 % openssl verify -CAfile ca-certificates.crt -untrusted interme
diates-all.pem www-chula-ac-th.cert

www-chula-ac-th.cert: OK
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 % openssl verify -CAfile ca-certificates.crt -untrusted interme
diates-all.pem classdeedee-cloud-cp-eng-chula-ac-th.cert
classdeedee-cloud-cp-eng-chula-ac-th.cert: OK
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 %
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 % python verify_chain.py \
  --batch twitter-com.cert google-com.cert www-chula-ac-th.cert classdeedee-cloud-cp-eng-chula-ac-th.cert \
  --intermediate intermediates-all.pem \
  --roots ca-certificates.crt
[twitter-com.cert] ✅ Certificate verified
[google-com.cert] ✅ Certificate verified
[www-chula-ac-th.cert] ✅ Certificate verified
[classdeedee-cloud-cp-eng-chula-ac-th.cert] ✅ Certificate verified
(datascience) pupipatsingkhorn@Pupipats-MacBook-Air activity-05 %
```

```
#!/usr/bin/env python3
"""
Q10 - Certificate chain validation (PKI Activity)
Requires:
  pip install pyopenssl pem

Files expected in the working directory:
  - target.cert            (leaf certificate in PEM; e.g., twitter_com.cert)
  - intermediate.cert      (one or more intermediates concatenated in PEM; optional but recommended)
  - ca-certificates.crt    (root CA bundle in PEM)

Usage examples:
  python verify_chain.py --target twitter_com.cert --intermediate intermediate.cert --roots ca-certificates.crt
  python verify_chain.py --batch twitter_com.cert google.cert chula.cert classdeedee.cert --intermediate intermediate.cert --roots ca-certificates.crt
"""
import argparse
import sys
from OpenSSL import crypto
import pem
from typing import List, Optional

def load_pem_file(path: str) -> str:
    with open(path, 'r', encoding='utf-8') as f:
        return f.read()

def parse_pem_bundle(path: str) -> List[str]:
    """Return a list of PEM-encoded certs as strings from a bundle file (e.g., ca-certificates.crt)."""
    pems = pem.parse_file(path)
    return [str(p) for p in pems]

def verify_chain_of_trust(cert_pem: str, trusted_cert_pems: List[str]) -> Optional[str]:
    """
    Returns None if validation succeeds (PyOpenSSL semantics), otherwise returns a string error message.
    """
    try:
        certificate = crypto.load_certificate(crypto.FILETYPE_PEM, cert_pem)
    except Exception as e:
        return f"Failed to parse target certificate: {e}"

    store = crypto.X509Store()
    for trusted_cert_pem in trusted_cert_pems:
        try:
            trusted_cert = crypto.load_certificate(crypto.FILETYPE_PEM, trusted_cert_pem)
            store.add_cert(trusted_cert)
        except Exception as e:
            return f"Failed to load a trusted certificate: {e}"

    try:
        store_ctx = crypto.X509StoreContext(store, certificate)
        result = store_ctx.verify_certificate()  # None on success
        return result  # None means OK
    except Exception as e:
        return str(e)

def main():
    parser = argparse.ArgumentParser(description="Verify an X.509 certificate chain of trust.")
    parser.add_argument('--target', help="PEM file for the leaf certificate to verify (e.g., twitter_com.cert).")
    parser.add_argument('--intermediate', help="PEM file containing intermediate cert(s). Optional but recommended.")
    parser.add_argument('--roots', default="ca-certificates.crt", help="Root CA bundle in PEM (default: ca-certificates.crt).")
    parser.add_argument('--batch', nargs='*', help="Verify multiple target certs. Provide file paths after --batch.")
    args = parser.parse_args()

    # Prepare trusted set: all roots (+ intermediates if provided)
    trusted: List[str] = []
    try:
        trusted.extend(parse_pem_bundle(args.roots))
    except Exception as e:
        print(f"[ERROR] Could not read root bundle '{args.roots}': {e}")
        sys.exit(2)

    if args.intermediate:
        try:
            # Allow multiple intermediates concatenated in one file
            trusted.extend(parse_pem_bundle(args.intermediate))
        except Exception:
            # Fallback: treat it as a single PEM (not a bundle parsed by pem)
            try:
                trusted.append(load_pem_file(args.intermediate))
            except Exception as e:
                print(f"[ERROR] Could not read intermediate cert(s) '{args.intermediate}': {e}")
                sys.exit(2)

    def verify_one(path: str) -> int:
        try:
            cert_pem = load_pem_file(path)
        except Exception as e:
            print(f"[{path}] ❌ Could not read: {e}")
            return 2

        err = verify_chain_of_trust(cert_pem, trusted)
        if err is None:
            print(f"[{path}] ✅ Certificate verified")
            return 0
        else:
            print(f"[{path}] ❌ Verification failed: {err}")
            # Common hint for 'unable to get local issuer certificate'
            if "unable to get local issuer" in err.lower() or "unable to get issuer" in err.lower():
                print("  Hint: Ensure all required intermediate certs are included via --intermediate.")
            return 1

    rc = 0
    if args.batch:
        for p in args.batch:
            rc |= verify_one(p)
    elif args.target:
        rc = verify_one(args.target)
    else:
        print("Provide --target <file> or use --batch <file1> <file2> ...")
        rc = 2
    sys.exit(rc)

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



```
#!/usr/bin/env bash
# get_site_certs.sh
# Save leaf and intermediate certs for a list of hosts.
# Output per host:
#   <host-with-dashes>.cert
#   <host-with-dashes>-intermediate.pem

set -euo pipefail

HOSTS=(
  "twitter.com"
  "google.com"
  "www.chula.ac.th"
  "classdeedee.cloud.cp.eng.chula.ac.th"
)

sanitize() {
  # replace anything not [A-Za-z0-9.-] with '-', collapse repeats, trim ends
  echo "$1" | sed -e 's/[^A-Za-z0-9.]/-/g' -e 's/-\{2,\}/-/g' -e 's/^-//' -e 's/-$//'
}

save_leaf_and_chain() {
  local host="$1" chainfile="$2"
  # Capture *all* certificates sent by the server
  openssl s_client -connect "${host}:443" -servername "${host}" -showcerts </dev/null 2>/dev/null \
    | awk '/BEGIN CERTIFICATE/,/END CERTIFICATE/{print}' > "${chainfile}"
}

extract_leaf() {
  local chainfile="$1" leaf_out="$2"
  # First PEM block = leaf
  awk 'BEGIN{f=0} /BEGIN CERTIFICATE/{c++} c==1 {print} /END CERTIFICATE/ && c==1 {exit}' \
    "${chainfile}" > "${leaf_out}"
}

extract_intermediates_from_chain() {
  local chainfile="$1" inter_out="$2"
  : > "${inter_out}"
  # All PEM blocks after the first = intermediates
  awk '
    /BEGIN CERTIFICATE/ {i++}
    i>=2 {print}
  ' "${chainfile}" >> "${inter_out}"
}

fetch_intermediates_via_aia_if_needed() {
  local host="$1" leaf="$2" inter_out="$3"
  # If we already have some intermediates from handshake, keep them; only add via AIA if empty
  if grep -q 'BEGIN CERTIFICATE' "${inter_out}" 2>/dev/null; then
    return 0
  fi

  echo "==> [${host}] no intermediates in handshake; trying AIA"
  # Pull CA Issuers URLs from the leaf; strip the leading "URI:" token if present
  while IFS= read -r url; do
    url="${url#URI:}"
    [ -n "$url" ] || continue
    tmp="$(mktemp)"
    if curl -fsSL "${url}" -o "${tmp}"; then
      # If DER, convert to PEM; else append as-is
      if openssl x509 -inform der -in "${tmp}" -outform pem >> "${inter_out}" 2>/dev/null; then
        :
      else
        cat "${tmp}" >> "${inter_out}"
      fi
    fi
    rm -f "${tmp}"
  done < <(
    openssl x509 -in "${leaf}" -noout -text \
    | awk '
        /Authority Information Access/ {inAIA=1; next}
        inAIA && /CA Issuers - URI:/ {print $NF; next}
        inAIA && /^[^ ]/ {inAIA=0}
      '
  )
}

main() {
  command -v openssl >/dev/null || { echo "OpenSSL not found"; exit 1; }
  command -v curl    >/dev/null || { echo "curl not found"; exit 1; }

  for h in "${HOSTS[@]}"; do
    base="$(sanitize "${h}")"              # e.g., google.com -> google-com
    chain="${base}-chain.pem"
    leaf="${base}.cert"                    # e.g., google-com.cert
    inter="${base}-intermediate.pem"       # e.g., google-com-intermediate.pem

    echo "==> [${h}] grabbing chain"
    save_leaf_and_chain "${h}" "${chain}"

    echo "==> [${h}] saving leaf -> ${leaf}"
    extract_leaf "${chain}" "${leaf}"
    if ! grep -q "BEGIN CERTIFICATE" "${leaf}"; then
      echo "!! [${h}] failed to save leaf cert"; rm -f "${chain}"; continue
    fi

    echo "==> [${h}] saving intermediates -> ${inter}"
    : > "${inter}"
    extract_intermediates_from_chain "${chain}" "${inter}"
    fetch_intermediates_via_aia_if_needed "${h}" "${leaf}" "${inter}"

    # Optional: quick subjects preview (comment out if you want *only* files)
    echo "   leaf:         $(openssl x509 -in "${leaf}" -noout -subject | sed 's/subject= //')"
    if grep -q "BEGIN CERTIFICATE" "${inter}"; then
      echo "   intermediates:"
      openssl crl2pkcs7 -nocrl -certfile "${inter}" | openssl pkcs7 -print_certs -noout | grep "subject=" | sed 's/subject= /      - /'
    else
      echo "   intermediates: (none found)"
    fi

    # Keep the chain file for debugging; delete if you don't want it:
    # rm -f "${chain}"
    echo
  done
}

main "$@"
```

Q11.

Nowaday, there are root certificates for class 1 and class 3. What uses would a class 1 signed certificate have that a class 3 doesn't, and vice versa?

Answer:

- Class 1 (lower-assurance): Typically used for individual / personal identity and low-risk uses such as S/MIME email, basic client authentication, or simple DV-style identity where only control of an address/account is checked. Advantage: quick issuance with minimal vetting; good for personal email signing/encryption.
- Class 3 (higher-assurance): Used for organization-level trust like TLS server authentication for public websites, code signing, and scenarios requiring stronger identity verification (org validation, stricter CP/CPS). Advantage: stronger vetting and liability statements, suitable for production web servers and software publishers.

This maps to the course emphasis that PKI trust depends on the CA's policy and verification rigor (CP/CPS) and on chaining to widely trusted roots for public services. Class 3 roots back higher-assurance use cases, while Class 1 roots are acceptable for lower-assurance/personal use.

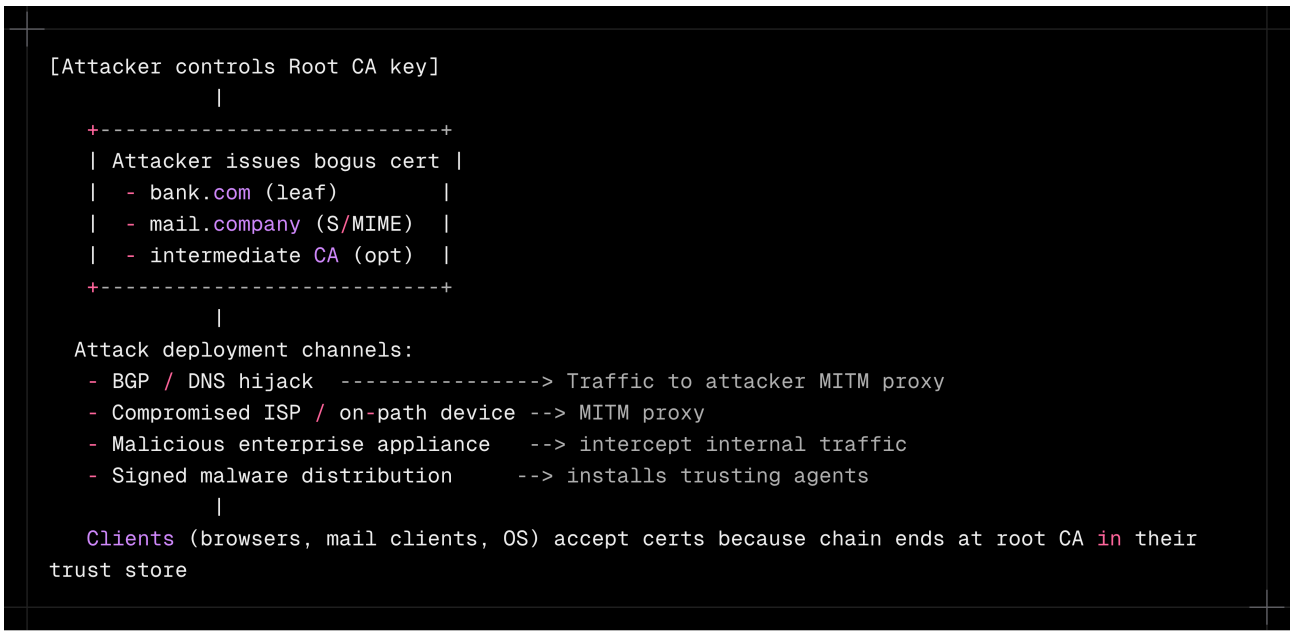
Q12.

Assuming that a Root CA in your root store is hacked and under the control of an attacker, and this is not noticed by anyone for months.

a. What further attacks can the attacker stage? Draw a possible attack setup.

Answer:

If a root CA is compromised and unnoticed for months, the attacker can issue valid certificates for any domain, create subordinate CAs, perform large-scale MITM (via DNS/BGP hijacks or on-path proxies), sign malware/firmware and S/MIME messages, and run phishing or impersonation campaigns, all appearing valid to clients because the chain terminates at a trusted root.



b. In the attack you have described above, can we rely on CRLs or OCSP for protection? Please explain.

Answer:

CRLs and OCSP alone cannot be relied on to protect against a root compromise. If the root or CA infrastructure is under attacker control, the attacker can block revocation checks, serve forged "good" OCSP responses, or simply not publish CRLs; many clients also soft-fail revocation checks for availability reasons. Stronger defenses include rapid detection (Certificate Transparency), vendor root-distrust updates, OCSP stapling with must-staple, short-lived certs, and multi-layer monitoring.