Class Test 2 NetSec

Alice and Bob participate in a public-key infrastructure that enables them to exchange legally binding digital signatures.

(i) Name two reasons why, for some purposes, Alice might prefer to use a message authentication code, instead of a digital signature, to protect the integrity and authenticity of her messages to Bob.

(ii) Outline a protocol for protecting the integrity and authenticity of Alice's messages to Bob that combines the benefits of a public-key infrastructure with those of using a message authentication code.

(i) Why Alice Might Prefer a MAC Over a Digital Signature

Alice might prefer a **Message Authentication Code (MAC)** instead of a digital signature for the following reasons:

1️⃣ Efficiency (Performance)

- **MACs are much faster** than digital signatures.
- They use symmetric cryptography (e.g., HMAC with SHA-256), which is computationally lightweight.
- Digital signatures (e.g., RSA, ECDSA) involve expensive asymmetric operations.

This matters when:

- Sending large volumes of messages
- Working on constrained devices
- Needing low latency communication

2️⃣ Lower Computational and Communication Overhead

- MACs generate **shorter tags** (e.g., 256 bits).
- Digital signatures are much larger (e.g., 2048-bit RSA signatures).
- MAC generation and verification consume fewer resources.

This is important in:

- High-throughput systems
- Embedded systems
- Real-time communication

3️⃣ Confidentiality of Authentication Key

- In MAC systems, only Alice and Bob share the secret key.
- No third party can verify or misuse authentication information.
- Digital signatures provide public verifiability, which may be undesirable if:
  - Alice does not want third parties to verify messages
  - She wants plausibly deniable communication

(ii) Protocol Combining PKI and MAC (Hybrid Approach)

We want:

- The efficiency of MACs
- The trust infrastructure of a Public-Key Infrastructure (PKI)

This is exactly how modern secure communication systems (like TLS) work.

🔐 Hybrid Protocol Outline

Assume:

- Alice and Bob each have public/private key pairs certified by a trusted CA.
- They know each other's certified public keys.

Step 1: Authentication Using Digital Signatures

1. Alice generates a random session key K.
2. Alice encrypts K using Bob's public key.
3. Alice signs the encrypted session key with her private key.
4. Alice sends:
   - Encrypted session key
   - Her certificate
   - Her digital signature

Bob:

- Verifies Alice's certificate (via CA).
- Verifies Alice's signature.
- Decrypts the session key using his private key.

Now both share secret session key K.

Step 2: Efficient Message Protection Using MAC

For each message **M**:

Alice sends:

```
nginx

M || MAC_K(M)
```

Where:

- **MAC_K(M)** is a message authentication code computed with session key **K**.

Bob:

- Recomputes MAC using **K**
- Verifies integrity and authenticity

✅ Security Properties Achieved

Property	How It Is Achieved
Authentication	PKI verifies identity initially
Integrity	MAC ensures message is unchanged
Efficiency	Only one expensive public-key operation
Scalability	PKI handles identity management
Forward secrecy (if extended with DH)	Possible with ephemeral keys

---

🎯 Why This Is Better Than Pure Digital Signatures

If Alice signed every message:

- High computational cost
- Large signatures
- Slower communication

Instead:

- Use **digital signatures once** (for key establishment)
- Use **MACs for all data**

💡 Real-World Example

This hybrid approach is used in:

- SSL/TLS
- IPsec

- Secure messaging protocols

They use public-key cryptography to set up secure session keys and symmetric cryptography (MACs + encryption) for actual data transfer.

If you'd like, I can rewrite this in short exam-ready 6–8 mark format.

> 2. Explain briefly mechanisms that software on a desktop computer can use to securely generate secret keys for use in cryptographic protocols.

To securely generate secret keys on a desktop computer, software must ensure **high entropy**, **unpredictability**, and **cryptographic strength**. Below are the main mechanisms used:

1️⃣ Use of Cryptographically Secure Pseudo-Random Number Generators (CSPRNGs)

A **CSPRNG** generates unpredictable random values suitable for cryptographic use.

How it works:

- Seeded with high-entropy input from the operating system.
- Uses secure algorithms (e.g., hash-based or block-cipher-based DRBGs).
- Produces random bits indistinguishable from true randomness.

Examples:

- `/dev/random`, `/dev/urandom` (Linux)
- `CryptGenRandom` / `BCryptGenRandom` (Windows)
- `SecRandomCopyBytes` (macOS)

These are much safer than standard PRNGs like `rand()`.

2️⃣ Collecting Entropy from Physical Sources

Desktop systems gather entropy from unpredictable physical events, such as:

- Keyboard timing
- Mouse movement timing
- Disk I/O timing
- Network packet timing

- Hardware interrupts

This randomness is collected into an **entropy pool**, which seeds the CSPRNG.

3️⃣ Hardware Random Number Generators (HRNGs)

Modern CPUs may include hardware-based randomness sources.

Examples:

- Intel `RDRAND` instruction
- TPM (Trusted Platform Module)
- Dedicated hardware noise generators

These rely on physical processes (thermal noise, electrical noise) and are often mixed into the OS entropy pool.

4️⃣ Key Derivation Functions (KDFs)

If keys are derived from passwords or passphrases, software must use secure **KDFs**:

Examples:

- PBKDF2
- Argon2
- scrypt
- HKDF

These:

- Stretch weak input into strong keys
- Add salt (prevents rainbow table attacks)
- Apply many iterations (resists brute force)

5️⃣ Proper Seeding and Reseeding

Secure systems:

- Properly seed the CSPRNG at boot.
- Periodically reseed it with fresh entropy.
- Protect internal state from exposure.

If the seed is predictable, all generated keys become predictable.

6️⃣ Use of Operating System APIs (Best Practice)

Applications should:

- NEVER implement their own RNG.
- ALWAYS use OS-provided cryptographic APIs.
- Let the OS manage entropy collection and mixing.

This prevents common mistakes like:

- Using timestamps as seeds
- Using non-cryptographic PRNGs

Summary

Secure key generation on desktops relies on:

- High-entropy physical randomness
- OS-managed entropy pools
- Cryptographically secure PRNGs
- Hardware randomness support
- Proper key derivation when needed

The fundamental requirement is **unpredictability** — if an attacker can predict the random numbers, they can predict the keys.

> Explain SSH keystroke timing attack. How does SSH defend against keystroke timing attacks?

🔐 SSH Keystroke Timing Attack

A **keystroke timing attack** against **SSH (Secure Shell)** exploits the fact that even though SSH encrypts the contents of communication, it may still leak **timing information** about when keys are pressed.

The attacker does **not** see the actual characters typed. Instead, they observe:

- The time interval between successive packets
- Packet sizes (especially small packets per keystroke)
- Patterns in typing rhythm

Because SSH typically sends data immediately after each keypress (interactive mode), each keystroke can generate a network packet.

🎯 How the Attack Works

1️⃣ Observation

An attacker (e.g., someone monitoring network traffic) passively observes SSH traffic and records:

- Timestamps of packets
- Packet lengths
- Direction of communication

Even though packets are encrypted, timing metadata is visible.

2️⃣ Timing Leakage

Human typing behavior has characteristics:

- Different keys take different time gaps.
- Digraph timing (time between two specific keys) varies.
- Password typing often has identifiable patterns.
- Backspace behavior reveals mistakes.
- Short commands like `ls`, `sudo`, `exit` have known structures.

By analyzing timing intervals, attackers may infer:

- Command length
- Specific commands
- Password characteristics
- Typing behavior of a specific user (biometric identification)

3️⃣ Practical Attacks

Research has shown attackers can:

- Guess short commands with high probability
- Infer password length
- Recognize repeated commands
- Identify specific users by typing patterns (behavioral fingerprinting)

The attack is called a **side-channel attack**, because it does not break encryption — it exploits leaked metadata.

🛡️ How SSH Defends Against Keystroke Timing Attacks

Modern SSH implementations (e.g., OpenSSH) include protections.

1️⃣ Packet Padding

SSH can add **random padding** to packets so:

- Packet sizes no longer directly correspond to keystrokes.
- Observers cannot easily infer message length.

2️⃣ Keystroke Obfuscation (Timing Obfuscation)

OpenSSH introduced a mitigation mechanism that:

- Inserts **dummy packets**
- Sends packets at more regular intervals
- Adds artificial delays

This makes real keystrokes harder to distinguish from noise.

Instead of:

```
css

Packet → (500 ms gap) → Packet → (200 ms gap) → Packet
```

It may look like:

```
csharp

Packet → (fixed 50 ms intervals with dummy packets)
```

This hides human timing patterns.

3️⃣ Interactive vs Bulk Mode Handling

For bulk data (e.g., file transfer via SCP/SFTP):

- Data is buffered and sent in larger chunks.
- Individual keystrokes do not directly generate packets.

This reduces timing resolution.

4️⃣ Encryption of Packet Length

SSH encrypts packet payloads and includes random padding inside encrypted packets. However:

- The outer packet size and timing are still visible.
- That is why additional defenses are needed.

5️⃣ Rate Limiting / Delayed Output

Some defenses involve:

- Delaying echo of typed characters
- Sending data in fixed-size frames

This reduces correlation between user input and packet timing.

⚠️ Why This Attack is Still Difficult

For the attack to succeed:

- The attacker must observe traffic very precisely.
- Network jitter can mask timing differences.
- TCP buffering and retransmissions distort patterns.
- Modern SSH implementations reduce leakage significantly.

So, it is **theoretically possible but practically challenging**.

🧠 Summary

What the attack exploits:

- Timing metadata
- Packet size
- Human typing rhythms

What SSH uses as defense:

- Packet padding
- Dummy packets
- Timing obfuscation
- Buffered transmission

The key idea:
Even when encryption is strong, **metadata leakage can still reveal sensitive information**, which is why modern systems defend against side-channel attacks like keystroke timing analysis.

What is a web application side channel attack? How do you defend against the same

🌐 Web Application Side-Channel Attack

A **web application side-channel attack** is an attack where an adversary learns sensitive information **indirectly** by observing unintended signals ("side channels") produced by a web application — instead of breaking its cryptography or authentication directly.

The attacker exploits **observable differences** such as:

- Response time
- Response size
- Error messages
- HTTP status codes
- Caching behavior
- Power/CPU usage (in advanced cases)

Even if the application is using HTTPS and strong cryptography, it may still leak information through these side effects.

🎯 Common Types of Web Side-Channel Attacks

1️⃣ Timing Attacks

The attacker measures how long the server takes to respond.

Example:

Password verification implemented like this:

```vbnet
compare input_password with real_password character by character
stop on first mismatch
```

If the first character is correct → slightly longer response time.
If first two are correct → slightly longer again.

By measuring timing differences, attacker can recover the password **one character at a time**.

2️⃣ Response Size Leakage

Even if content is encrypted via HTTPS, attackers may observe:

- Different response lengths
- Different compression sizes (CRIME/BREACH attack)
- Different page content lengths

Example:

- "User exists" → 500 bytes
- "User does not exist" → 450 bytes
  → Username enumeration possible.

3️⃣ Error Message Differences

Different error responses can reveal:

- Whether username is valid
- Whether password is partially correct
- Whether account is locked

Example:

- "Invalid username"
- "Invalid password"
  → Reveals valid usernames.

4️⃣ Cache-based Attacks

Attackers observe:

- Whether a resource loads faster from cache
- Cross-site timing differences
- Browser cache probing

Can reveal:

- Whether user visited a certain site
- Whether user is logged in

5️⃣ CPU / Resource Usage Attacks

If an operation consumes more CPU for certain inputs:

- Complex SQL query vs simple one

- Valid token vs invalid token
  Timing difference reveals secret structure.

🛡️ How to Defend Against Web Side-Channel Attacks

1️⃣ Constant-Time Implementations

Critical comparisons must be done in constant time.

Instead of:

```kotlin
kotlin

if (a[i] != b[i]) return false
```

Use:

- Constant-time comparison functions
- Cryptographic library safe comparisons

This prevents timing-based secret leakage.

2️⃣ Uniform Error Messages

Always return the same response for authentication failures.

Instead of:

```nginx
nginx

Invalid username
Invalid password
```

Return:

```nginx
nginx

Invalid credentials
```

Avoid giving attackers clues.

3️⃣ Normalize Response Time

Ensure similar execution time regardless of input.

Techniques:

- Add fixed delay

- Equalize code paths
- Pad responses

4️⃣ Normalize Response Size

Ensure:

- Same content length for success/failure
- Same HTTP status codes if possible
- Disable compression when needed (to prevent compression attacks like BREACH)

5️⃣ Rate Limiting & Monitoring

Side-channel attacks often require many repeated measurements.

Defenses:

- Rate limit login attempts
- Detect abnormal request patterns
- Use CAPTCHA for suspicious activity

6️⃣ Proper Session Management

Avoid leaks through:

- Predictable tokens
- Token validation timing
- Session ID patterns

Use:

- Strong randomness
- Secure cookies (HttpOnly, Secure, SameSite)

7️⃣ Disable Unnecessary Information Leaks

Avoid exposing:

- Stack traces
- Internal error details
- SQL errors
- Debug messages

Use proper exception handling.

🧠 Key Insight

Side-channel attacks do **not** break encryption or passwords directly.

They exploit:

> Information leakage through metadata and unintended behavior.

Even if:

- HTTPS is enabled
- Strong cryptography is used

An application can still leak secrets via:

- Timing
- Size
- Errors
- Cache behavior

✅ Summary

Web application side-channel attack = extracting secrets by observing indirect signals (timing, size, errors).

Defenses include:

- Constant-time comparisons
- Uniform responses
- Response size normalization
- Rate limiting
- Monitoring
- Secure coding practices

Strong cryptography alone is not enough — implementation must also avoid leaking side-channel information.