# Digital Signatures

## MAT364 - Cryptography Course

**Instructor:** Adil Akhmetov

**University:** SDU

**Week 9**

Press Space for next page →

# What Are Digital Signatures?

## Purpose

- **Authenticity**: Proves who created the message
- **Integrity**: Detects any tampering
- **Non-repudiation**: Signer cannot deny the signature

## Building Blocks

- Cryptographic hash (e.g., SHA-256)
- Asymmetric keys (RSA, ECDSA, Ed25519)
- Secure padding or deterministic nonce generation

## High-Level Flow

1. Compute hash of message m → h = H(m)
2. Sign h with private key → σ
3. Verify with public key: Verify(m, σ) → true/false

## Real-World Uses

- Software/code signing
- TLS certificates and OCSP
- Package registries (npm, PyPI)
- Documents and PDFs

**Key idea:** Sign the hash, not the raw message; verify with the corresponding public key.

# RSA-PSS Signatures

# RSA-PSS: Secure Padding for Signatures

## Why PSS?

- Textbook RSA signatures are malleable and insecure
- **PSS (Probabilistic Signature Scheme)** provides provable security
- Random salt per signature → protects against structure-based attacks

## Guidance

- Use 2048–3072-bit RSA keys
- Hash: SHA-256 or stronger
- Public exponent e = 65537

## Python Example ( `cryptography` )

```python
from cryptography.hazmat.primitives.asymmetric import r
from cryptography.hazmat.primitives import hashes

# Key generation
private_key = rsa.generate_private_key(public_exponent=
public_key = private_key.public_key()

message = b"Approve PR #42"

# Sign (RSA-PSS)
signature = private_key.sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
```

# ECDSA and Ed25519

# ECDSA: Elliptic Curve Signatures

## Concepts

- Works over elliptic curves (e.g., secp256r1)
- Requires a fresh, uniformly random nonce k for each signature
- If k repeats or leaks → private key recovery

## Deterministic k (RFC 6979)

- Derive k from (privkey, H(m)) via HMAC-DRBG
- Avoids bad RNG failures

## Security

- Comparable security to RSA with much smaller keys
- Widely used in TLS, JWT libraries, blockchain systems

## Python Example (ECDSA)

```
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import hashes

private_key = ec.generate_private_key(ec.SECP256R1())
public_key = private_key.public_key()
msg = b"pay 10.00 USD"

signature = private_key.sign(msg, ec.ECDSA(hashes.SHA25
public_key.verify(signature, msg, ec.ECDSA(hashes.SHA25
```

## Ed25519 (EdDSA)

- Twisted Edwards curve; deterministic and fast
- Safer API: no need to manage nonces

```
from cryptography.hazmat.primitives.asymmetric import e

sk = ed25519.Ed25519PrivateKey.generate()
pk = sk.public_key()
sig = sk.sign(b"hello")
```

# Hash-Then-Sign and Pitfalls

## Correct Pattern

1. Canonicalize/serialize the message
2. Compute h = H(m)
3. Sign h using the scheme's API (or pass m if API handles hashing)

## Do/Don't

- Do: include domain separation/context strings
- Do: sign exactly what you intend to verify
- Don't: sign non-canonical JSON or ambiguous encodings
- Don't: mix encodings (UTF-8 vs UTF-16) without care

## Common Vulnerabilities

- Nonce reuse in ECDSA (leaks private key)
- Using raw RSA (no PSS) → forgery
- Malleable or ambiguous message formats
- Ignoring verification errors or exceptions

## Mitigations

- Prefer Ed25519 or ECDSA with RFC 6979
- Use RSA-PSS for RSA signatures
- Define a canonical serialization (e.g., CBOR/JSON canonical)

# 🎯 Student Task: Verify and Break

## Part A: Verify a Signature

Given `message = "invoice #1234: 250 USD"`, generate an Ed25519 key pair, sign the message, and verify it.

## Part B: Spot the Bug

Consider ECDSA signing that uses a fixed `k = 42` for all messages. Explain how an attacker recovers the private key from two signatures over different messages.

### Deliverables

- Ed25519 signature (hex) and verification result
- Short explanation of the ECDSA nonce reuse attack

# ✅ Solution Sketch

## Part A (Ed25519)

```python
from cryptography.hazmat.primitives.asymmetric import ed25519

msg = b"invoice #1234: 250 USD"
sk = ed25519.Ed25519PrivateKey.generate()
pk = sk.public_key()
sig = sk.sign(msg)

try:
    pk.verify(sig, msg)
    print("valid")
except Exception:
    print("invalid")
```

## Part B (Nonce Reuse in ECDSA)

If the same nonce k is reused for two signatures (r, s1) on m1 and (r, s2) on m2 over the same key, then an attacker computes k = (H(m1) - H(m2)) / (s1 - s2) mod n and recovers the private key d from the signature equation. Hence, k must be unique and unpredictable (or derived deterministically per RFC 6979).

# Real-World Applications

## Code and Package Signing

- OS and driver signing (Windows, macOS, Linux distributions)
- Package managers: signed manifests and release keys
- Supply-chain security (Sigstore, Cosign)

## Documents and Protocols

- PDF/XML signing; long-term validation (timestamps)
- JWT/JWS for API authentication (ES256, RS256, EdDSA)
- Secure updates and firmware signing

## Web PKI

- X.509 certificates bind domain names to public keys
- Certificate Transparency logs
- OCSP stapling

## Recommendations

- Prefer Ed25519 or ECDSA (P-256) for new systems
- Use RSA-PSS where RSA is required
- Use robust, vetted libraries; never roll your own crypto

# Best Practices

## Design and Implementation

- Canonicalize data before signing
- Include context strings and versioning in the signed payload
- Store public keys and metadata securely
- Rotate keys; support key revocation

## Operational Security

- Protect private keys with HSMs or secure enclaves
- Enforce strong randomness or deterministic nonces
- Log verification results and failures
- Test negative cases (tampering should fail)

# Public Key Infrastructure (PKI)

# Certificate Chains and Trust

## Certificate Hierarchy

- **Root CA** - Self-signed, trusted by OS/browser
- **Intermediate CA** - Signed by root, signs end-entities
- **End-entity** - Server certificates, code signing certs

## Trust Model

- **Web PKI** - Browser trust stores (Mozilla, Microsoft, Apple)
- **Enterprise PKI** - Internal CAs for corporate networks
- **Code signing** - Microsoft, Apple, Google trusted roots

## Certificate Structure (X.509)

```
Certificate:
  Version: 3
  Serial Number: unique identifier
  Issuer: CA that signed this cert
  Subject: entity this cert belongs to
  Validity: notBefore, notAfter
  Public Key: RSA/ECC public key
  Extensions: SAN, Key Usage, etc.
  Signature: CA's signature over above fields
```

## Key Extensions

- **Subject Alternative Name (SAN)** - Multiple domains/IPs
- **Key Usage** - Digital signature, key encipherment, etc.
- **Extended Key Usage** - TLS server, code signing, etc.

# Certificate Validation Process

## Validation Steps

1. **Parse certificate** - Check format and version
2. **Check validity period** - Current time within notBefore/notAfter
3. **Verify signature** - CA's signature over certificate fields
4. **Build chain** - Trace back to trusted root
5. **Check revocation** - CRL or OCSP response
6. **Validate extensions** - Key usage, SAN, etc.

## Common Validation Libraries

- **OpenSSL** - C library, used by many applications
- **cryptography (Python)** - High-level Python bindings
- **Bouncy Castle** - Java/C# cryptographic library

## Python Certificate Validation

```
from cryptography import x509
from cryptography.x509.verification import PolicyBuilde
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import r
import datetime

# Load certificate
with open("server.crt", "rb") as f:
    cert_data = f.read()
cert = x509.load_pem_x509_certificate(cert_data)

# Basic validation
now = datetime.datetime.now()
if now < cert.not_valid_before or now > cert.not_valid_
    raise ValueError("Certificate expired")
```

# JSON Web Tokens (JWT)

# JWT Structure and Signing

## JWT Components

```
Header.Payload.Signature
```

**Header:**

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

**Payload:**

```
{
  "sub": "user123",
  "iat": 1516239022,
  "exp": 1516242622,
  "iss": "https://auth.example.com"
}
```

## Supported Algorithms

## JWT Implementation

```
import jwt
from cryptography.hazmat.primitives import serializatio
from cryptography.hazmat.primitives.asymmetric import r

# Generate RSA key pair
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)
public_key = private_key.public_key()

# Create JWT
payload = {
    "sub": "user123",
    "iat": 1516239022
```

# JWT Security Considerations

## Common Vulnerabilities

- **Algorithm confusion** - "none" algorithm attack
- **Weak secrets** - Predictable HMAC keys
- **Key confusion** - Using public key as HMAC secret
- **Timing attacks** - Signature verification timing

## Best Practices

- **Validate algorithm** - Only accept expected algorithms
- **Short expiration** - Limit token lifetime
- **Secure storage** - Protect private keys
- **Key rotation** - Regular key updates

## Secure JWT Implementation

```
import jwt
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import P
import secrets

class SecureJWT:
    def __init__(self, secret_key):
        self.secret_key = secret_key
        self.allowed_algs = ["HS256", "RS256", "ES256"]

    def create_token(self, payload, algorithm="HS256"):
        if algorithm not in self.allowed_algs:
            raise ValueError(f"Algorithm {algorithm} no

        # Add standard claims
```

# Blockchain and Cryptocurrency
# Signatures

# Bitcoin Transaction Signatures

## Bitcoin Transaction Structure

```
Transaction:
  Inputs: [previous_tx_hash, output_index, scriptSig]
  Outputs: [value, scriptPubKey]
  Locktime: block height or timestamp
```

## ScriptSig and ScriptPubKey

- **ScriptSig** - Contains signature and public key
- **ScriptPubKey** - Defines spending conditions
- **P2PKH** - Pay-to-Public-Key-Hash (most common)

## Signature Process

1. Create transaction hash (double SHA-256)
2. Sign hash with private key (ECDSA)
3. Include signature in ScriptSig
4. Miners verify signature during validation

## Bitcoin Signature Implementation

```python
import hashlib
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization impor

class BitcoinTransaction:
    def __init__(self, private_key):
        self.private_key = private_key
        self.public_key = private_key.public_key()

    def create_transaction_hash(self, inputs, outputs,
        """Create transaction hash for signing"""
        # Simplified - real implementation is more comp
        tx_data = f"{inputs}{outputs}{locktime}".encode
        return hashlib.sha256(hashlib.sha256(tx_data).
```

# Ethereum Smart Contract Signatures

## Ethereum Transaction Structure

```
Transaction:
  nonce: transaction sequence number
  gasPrice: fee per gas unit
  gasLimit: maximum gas to use
  to: recipient address
  value: ETH amount
  data: contract call data
  v, r, s: signature components
```

## EIP-1559 (London Fork)

- **Base fee** - Network-determined fee
- **Priority fee** - User-determined tip
- **Max fee** - Maximum user willing to pay

## Ethereum Signature Implementation

```python
import rlp
from eth_account import Account
from eth_account.messages import encode_defunct
from web3 import Web3

class EthereumTransaction:
    def __init__(self, private_key):
        self.private_key = private_key
        self.account = Account.from_key(private_key)

    def create_transaction(self, to, value, gas_price,
        """Create unsigned transaction"""
        return {
            'to': to,
            'value': value
```

# Advanced Signature Schemes

# Threshold Signatures

## Concept

- **Threshold (t,n)** - Any t out of n parties can sign
- **Secret sharing** - Private key split among parties
- **Distributed signing** - No single party has full key
- **Applications** - Multi-sig wallets, consensus protocols

## Benefits

- **Fault tolerance** - Works even if some parties fail
- **Security** - No single point of failure
- **Flexibility** - Adjustable threshold

## Simplified Implementation

```
from cryptography.hazmat.primitives.asymmetric import e
import secrets

class ThresholdSignature:
    def __init__(self, threshold, total_parties):
        self.threshold = threshold
        self.total_parties = total_parties
        self.shares = []

    def generate_shares(self):
        """Generate secret shares (simplified)"""
        # In practice, use Shamir's Secret Sharing
        master_key = ed25519.Ed25519PrivateKey.generate

        for i in range(self.total_parties):
```

# Blind Signatures

## Concept

- **Blind signature** - Signer doesn't see the message
- **Unlinkability** - Signer can't link signature to message
- **Applications** - Electronic voting, anonymous credentials

## Process

1. **Blinding** - User blinds message with random factor
2. **Signing** - Signer signs blinded message
3. **Unblinding** - User removes blind factor
4. **Verification** - Anyone can verify unblinded signature

## Blind Signature Implementation

```
from cryptography.hazmat.primitives.asymmetric import r
from cryptography.hazmat.primitives import hashes
import secrets

class BlindSignature:
    def __init__(self):
        self.private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )
        self.public_key = self.private_key.public_key()

    def blind_message(self, message, blinding_factor):
        """Blind the message"""
        n = self.public_key.public_numbers().n
```

# Signature Performance and Optimization

# Performance Comparison

## Algorithm Performance

| Algorithm | Key Size | Signature Size | Sign Speed | Verify Speed |
|---|---|---|---|---|
| RSA-2048 | 2048 bits | 256 bytes | Slow | Fast |
| RSA-3072 | 3072 bits | 384 bytes | Slower | Medium |
| ECDSA P-256 | 256 bits | 64 bytes | Fast | Fast |
| Ed25519 | 256 bits | 64 bytes | Very Fast | Very Fast |

## Benchmarking Code

```
import time
from cryptography.hazmat.primitives.asymmetric import r
from cryptography.hazmat.primitives import hashes

def benchmark_signature_algorithm(algorithm_name, priva
    """Benchmark signature algorithm performance"""

    # Signing benchmark
    start_time = time.time()
    for _ in range(iterations):
        if algorithm_name == "RSA":
            signature = private_key.sign(
                message,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256())
```

## When to Use What

# Hardware Security Modules (HSMs)

## HSM Benefits

- **Tamper resistance** - Physical protection
- **Key isolation** - Keys never leave HSM
- **Audit logging** - Track all operations
- **Compliance** - Meet regulatory requirements

## HSM Types

- **Network HSMs** - Remote access over network
- **USB HSMs** - Direct computer connection
- **Smart cards** - Portable, PIN-protected
- **Cloud HSMs** - AWS CloudHSM, Azure Dedicated HSM

## HSM Integration Example

```
from cryptography.hazmat.primitives.asymmetric import r
from cryptography.hazmat.primitives import serializatio
import pkcs11

class HSMKeyManager:
    def __init__(self, hsm_lib_path, token_label, pin):
        self.lib = pkcs11.lib(hsm_lib_path)
        self.token = self.lib.get_token(token_label)
        self.session = self.token.open(rw=True, user_pi

    def generate_key_pair(self, key_id, key_size=2048):
        """Generate RSA key pair in HSM"""
        public_key, private_key = self.session.generate
            pkcs11.Mechanism.RSA_PKCS_KEY_PAIR_GEN,
            pkcs11.KeyType.RSA
```

# Signature Attacks and Defenses

# Common Signature Attacks

## ECDSA Nonce Reuse

- **Problem** - Same k used for multiple signatures
- **Attack** - Recover private key from two signatures
- **Defense** - Use RFC 6979 deterministic nonces

## Fault Injection Attacks

- **Problem** - Hardware faults during signing
- **Attack** - Extract private key from faulty signatures
- **Defense** - Error checking, redundant computations

## Side-Channel Attacks

- **Problem** - Timing/power analysis reveals key
- **Attack** - Statistical analysis of execution traces
- **Defense** - Constant-time implementations

## Attack Mitigation Example

```
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import hashes
import hmac
import hashlib

class SecureECDSA:
    def __init__(self, private_key):
        self.private_key = private_key
        self.public_key = private_key.public_key()

    def deterministic_nonce(self, message_hash):
        """Generate deterministic nonce per RFC 6979"""
        # Simplified implementation
        private_bytes = self.private_key.private_bytes(
            encoding=serialization.Encoding.DER
```

# Post-Quantum Signatures

## Quantum Threat

- **Shor's algorithm** - Breaks RSA and ECDSA
- **Grover's algorithm** - Reduces symmetric key security
- **Timeline** - 10-30 years until practical quantum computers

## Post-Quantum Algorithms

- **Lattice-based** - Dilithium, Falcon
- **Hash-based** - SPHINCS+, XMSS
- **Code-based** - Classic McEliece
- **Multivariate** - Rainbow

## Dilithium Implementation (Concept)

```
# Conceptual implementation - real Dilithium is much mo
class DilithiumSignature:
    def __init__(self, security_level=2):
        self.security_level = security_level
        # Parameters vary by security level
        self.n = 256  # Polynomial degree
        self.q = 8380417  # Modulus
        self.eta = 2  # Secret coefficient bound

    def key_generation(self):
        """Generate Dilithium key pair"""
        # Simplified key generation
        # Real implementation uses polynomial arithmeti
        secret_key = self.generate_secret_polynomial()
        public_key = self.compute_public_key(secret_ke
```

# Questions?

Let's discuss digital signatures! ✍️

**Next Week:** Cryptographic protocols in web development (TLS usage, JWTs in practice).

**Assignment:** Implement Ed25519 signing/verification with canonical JSON and test tamper detection.