Key Exchange and Protocols

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 8

Press Space for next page →

The Key Exchange Problem

The Challenge

- **Alice and Bob** want to communicate securely
- **No pre-shared secret** exists
- **Eve eavesdrops** on all communication
- **How to establish** shared secret key?

Before Public Key Crypto

- **Courier delivery** - Slow and expensive
- **Pre-shared keys** - Doesn't scale
- **Trusted third party** - Single point of failure
- **Physical meeting** - Not always possible

The Breakthrough

Diffie-Hellman (1976):

- First practical key exchange protocol
- Based on discrete logarithm problem
- Allows secure key establishment over insecure channel
- Foundation for modern cryptography

Real-World Importance

- **TLS/SSL** - Secure web browsing
- **VPN** - Secure remote access
- **SSH** - Secure shell connections
- **Signal/WhatsApp** - Secure messaging

Key insight: Key exchange enables secure communication without pre-shared secrets!

Diffie-Hellman Key Exchange

Diffie-Hellman Protocol

How It Works

1. **Public parameters:** prime p and generator g
2. **Alice's secret:** random a, sends $A = g^a \bmod p$
3. **Bob's secret:** random b, sends $B = g^b \bmod p$
4. **Shared secret:** Both compute $s = g^{(ab)} \bmod p$

Mathematical Foundation

- **Discrete log problem:** Given g, p, and g^a, hard to find a
- **Diffie-Hellman problem:** Given g^a and g^b, hard to find $g^{(ab)}$
- **Security:** Relies on hardness of these problems

Example

```
# Public parameters
p = 23   # Prime modulus
g = 5    # Generator

# Alice's side
a = 6  # Alice's secret
A = pow(g, a, p)  # A = 5^6 mod 23 = 8

# Bob's side
b = 15  # Bob's secret
B = pow(g, b, p)  # B = 5^15 mod 23 = 19

# Shared secret computation
# Alice computes: s = B^a mod p = 19^6 mod 23 = 2
s_alice = pow(B, a, p)
```

DH Implementation

Complete Implementation

```
import secrets
from cryptography.hazmat.primitives import serializatic
from cryptography.hazmat.primitives.asymmetric import c

class DiffieHellmanKeyExchange:
    def __init__(self, key_size=2048):
        # Generate parameters (usually pre-shared)
        self.parameters = dh.generate_parameters(
            generator=2,
            key_size=key_size
        )

        # Generate private key
        self.private_key = self.parameters.generate_pri
```

Usage Example

```
# Alice's side
alice = DiffieHellmanKeyExchange()
alice_public = alice.get_public_bytes()

# Bob's side
bob = DiffieHellmanKeyExchange()
bob_public = bob.get_public_bytes()

# Exchange public keys (over insecure channel)
# ...

# Compute shared secrets
alice_shared = alice.compute_shared_secret(bob_public)
bob_shared = bob.compute_shared_secret(alice_public)
```

Man-in-the-Middle Attack

The Attack

Eve intercepts and modifies:

1. Alice sends A → Eve intercepts, sends E_1
2. Bob sends B → Eve intercepts, sends E_2
3. Alice thinks she's talking to Bob (key K_1)
4. Bob thinks he's talking to Alice (key K_2)
5. Eve can decrypt, read, re-encrypt all messages

Why It Works

- **No authentication** in basic DH
- **Eve controls** the channel
- **Alice and Bob** can't detect the attack
- **Need authentication** to prevent this

Attack Diagram

```
Alice              Eve              Bob
  |                 |                |
  ├─ A ⟶            |                |
  |                 ├─ E1 ⟶          |
  |                 |                |
  |                 |      ⟵ B ─┤
  |      ⟵ E2 ─┤                     |
  |                 |                |
K1 = E2^a       K1 = A^e1       K2 = E1^b
                K2 = B^e2
```

Solution: Authenticated DH

- **Sign** public keys with private key
- **Verify signatures** before computing shared secret
- **Use certificates** to bind identity to public key
- **Station-to-Station** protocol
- **TLS handshake** includes authentication

🎯 Student Task: DH Key Exchange

Task: Manual DH Calculation

Given public parameters:

- p = 71 (prime)
- g = 7 (generator)
- Alice's secret: a = 5
- Bob's secret: b = 12

Your tasks:

1. Calculate Alice's public value A = g^a mod p
2. Calculate Bob's public value B = g^b mod p
3. Calculate shared secret from Alice's perspective: s = B^a mod p
4. Calculate shared secret from Bob's perspective: s = A^b mod p
5. Verify they match

Bonus: Explain why Eve can't compute the shared secret even though she sees A and B.

✅ Solution: DH Calculation

Step-by-Step Solution

Given: $p = 71$, $g = 7$, $a = 5$, $b = 12$

Step 1: Alice's public value

```
A = g^a mod p = 7^5 mod 71 = 16807 mod 71 = 51
```

Step 2: Bob's public value

```
B = g^b mod p = 7^12 mod 71 = 13841287201 mod 71 = 4
```

Step 3: Alice computes shared secret

```
s = B^a mod p = 4^5 mod 71 = 1024 mod 71 = 30
```

Step 4: Bob computes shared secret

```
s = A^b mod p = 51^12 mod 71 = 30
```

Elliptic Curve Diffie-Hellman (ECDH)

ECDH Overview

Why ECDH?

- **Smaller keys** - 256-bit ECC ≈ 3072-bit RSA
- **Faster** - More efficient operations
- **Lower bandwidth** - Smaller key transmission
- **Mobile-friendly** - Less computation/storage

How It Works

1. **Agree on curve** and base point G
2. **Alice:** secret a, public A = aG
3. **Bob:** secret b, public B = bG
4. **Shared secret:** S = aB = bA = abG

Implementation

```
from cryptography.hazmat.primitives.asymmetric import e
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKD

class ECDHKeyExchange:
    def __init__(self):
        # Generate private key on secp256r1 curve
        self.private_key = ec.generate_private_key(ec.S
        self.public_key = self.private_key.public_key(

    def get_public_bytes(self):
        """Get public key for transmission"""
        return self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPu
```

TLS Handshake Protocol

TLS 1.3 Handshake

Handshake Flow

ClientHello:

- Supported cipher suites
- Key share (ECDHE public key)
- Supported groups (curves)

ServerHello:

- Selected cipher suite
- Key share (server's public key)
- Server certificate

Key Derivation:

- Both compute shared secret
- Derive encryption keys
- Start encrypted communication

Simplified Implementation

```
class TLSHandshake:
    def __init__(self, is_server=False):
        self.is_server = is_server
        self.ecdh = ECDHKeyExchange()
        self.cipher_suites = [
            'TLS_AES_256_GCM_SHA384',
            'TLS_CHACHA20_POLY1305_SHA256'
        ]

    def create_client_hello(self):
        """Create ClientHello message"""
        return {
            'version': 'TLS 1.3',
            'cipher_suites': self.cipher_suites,
            'key_share': self.ecdh.get_public_bytes()
```

Perfect Forward Secrecy (PFS)

What is PFS?

- **Ephemeral keys** for each session
- **Past sessions** remain secure even if long-term key compromised
- **No single key** can decrypt all past traffic
- **Essential** for long-term security

Without PFS

- RSA key exchange: same key encrypts many sessions
- If private key leaked → all past traffic decryptable
- "Decrypt all historical traffic" attack

With PFS (DHE/ECDHE)

- **New ephemeral key** for each session
- **Session keys** discarded after use
- **Past sessions** remain secure
- **TLS 1.3** mandates PFS

Implementation

```
class PFSSession:
    def __init__(self):
        # Generate ephemeral key pair
        self.ephemeral_key = ec.generate_private_key(ec

    def start_session(self, peer_public):
        """Start new session with PFS"""
        # Compute session key
        session_key = self.ephemeral_key.exchange(
            ec.ECDH(), peer_public
        )
```

Authenticated Key Exchange

Station-to-Station (STS) Protocol

Protocol Flow

1. **Alice → Bob:** g^a
2. **Bob → Alice:** g^b, Sig_B(g^a, g^b)
3. **Alice → Bob:** Sig_A(g^a, g^b)
4. **Both verify** signatures with certificates
5. **Compute** shared secret

Security Properties

- **Authentication** - Both parties verified
- **Key confirmation** - Both know they have same key
- **Perfect forward secrecy** - Ephemeral keys
- **Prevents MITM** - Signatures bind identity

Implementation Sketch

```
class STSProtocol:
    def __init__(self, private_key, certificate):
        self.private_key = private_key  # Long-term sig
        self.certificate = certificate
        self.dh = DiffieHellmanKeyExchange()

    def initiate(self):
        """Initiator sends DH public value"""
        return self.dh.get_public_bytes()

    def respond(self, initiator_public):
        """Responder sends DH public + signature"""
        my_public = self.dh.get_public_bytes()

        # Sign both public values
```

Signal Protocol (Double Ratchet)

Key Features

- **End-to-end encryption** for messaging
- **Perfect forward secrecy** for every message
- **Future secrecy** - Compromise doesn’t affect future
- **Used by** Signal, WhatsApp, Facebook Messenger

Ratchets

DH Ratchet:

- New ECDH key pair for each message exchange
- Provides forward secrecy

Symmetric Ratchet:

- Derives new keys from previous keys
- KDF chain for message keys

Simplified Concept

```
class DoubleRatchet:
    def __init__(self):
        self.dh_key = ec.generate_private_key(ec.SECP25
        self.root_key = None
        self.chain_key = None
        self.message_number = 0

    def dh_ratchet(self, peer_public):
        """Perform DH ratchet step"""
        # Compute new shared secret
        dh_output = self.dh_key.exchange(ec.ECDH(), pee

        # Derive new root and chain keys
        self.root_key, self.chain_key = self.kdf_rk(
            self.root_key, dh_output
```

Key Derivation Functions (KDFs)

Purpose

- **Transform** shared secret into usable keys
- **Expand** short secret into multiple keys
- **Extract** entropy from non-uniform sources
- **Domain separation** - Different keys for different purposes

HKDF (HMAC-based KDF)

- **Extract:** Extract fixed-length key from source
- **Expand:** Expand key to desired length
- **Standard:** RFC 5869

Implementation

```
from cryptography.hazmat.primitives.kdf.hkdf import HKI
from cryptography.hazmat.primitives import hashes

def derive_keys(shared_secret, salt=None):
    """Derive multiple keys from shared secret"""

    # Extract phase
    kdf_extract = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b'master'
    )
    master_key = kdf_extract.derive(shared_secret)
```

🎯 Student Task: Protocol Analysis

Task: Design Secure Protocol

Scenario: Alice and Bob want to establish a secure channel for exchanging messages.

Requirements:

1. Mutual authentication (both verify each other's identity)
2. Perfect forward secrecy
3. Resistance to man-in-the-middle attacks
4. Efficient for mobile devices

Your task:

1. Choose key exchange method (DH or ECDH)
2. Design authentication mechanism
3. Specify key derivation process
4. Describe message encryption scheme
5. Identify potential vulnerabilities

Deliverable: Protocol description with security analysis

✅ Solution: Protocol Design

Recommended Solution

Protocol: Authenticated ECDHE with Certificates

1. **Key Exchange:** ECDHE (secp256r1 or X25519)
2. **Authentication:** Digital signatures (ECDSA)
3. **Key Derivation:** HKDF-SHA256
4. **Encryption:** AES-256-GCM

Flow:

```
Alice → Bob: A = aG, Cert_A
Bob → Alice: B = bG, Cert_B, Sig_B(A || B)
Alice → Bob: Sig_A(A || B)
Both: Verify certs and signatures, compute K = abG
Both: Derive keys = HKDF(K, "encryption" || "mac")
```

Security:

- ECDHE provides PFS
- Certificates prevent MITM

Real-World Protocols

TLS 1.3

- **Mandatory PFS** (DHE/ECDHE only)
- **1-RTT** handshake
- **0-RTT** resumption (with replay risk)
- **Simplified** cipher suites

WireGuard VPN

- **Modern** cryptographic primitives
- **Simple** protocol design
- **Fast** performance
- **Noise Protocol** framework

SSH (Secure Shell)

- **Key exchange:** DH group exchange
- **Authentication:** Public key, password, certificates
- **Channel security:** Separate keys for each direction
- **Applications:** Remote login, file transfer

IPsec

- **IKEv2** key exchange
- **ESP/AH** protocols
- **SA (Security Association)** establishment
- **VPN** and network-level security

Best Practices

Protocol Design

- **Use established protocols** (TLS, Signal)
- **Avoid custom protocols** unless expert
- **Get security review** before deployment
- **Follow standards** (NIST, IETF RFCs)

Implementation

- **Use vetted libraries** (OpenSSL, cryptography)
- **Validate all inputs** and certificates
- **Implement timeouts** and limits
- **Test edge cases** thoroughly

Security Considerations

- **Always authenticate** key exchange
- **Use perfect forward secrecy**
- **Derive keys properly** (HKDF)
- **Check certificate validity**
- **Implement revocation** checking

Common Mistakes

- ❌ Unauthenticated DH
- ❌ Reusing ephemeral keys
- ❌ Weak random number generation
- ❌ Missing certificate validation
- ❌ Poor error handling

Questions?

Let's discuss key exchange protocols! 💬

Next Week: We'll explore digital signatures and their applications in detail.

Assignment: Implement ECDH key exchange with proper key derivation.