Implementation of Three Cryptographic Protocols

Applied Cryptography

25 september 2024

Hussaini Mohmmad Qasim, Volonterio Luca

Contents

1 Introduction

This lab focuses on the implementation of three fundamental cryptographic protocols in Python: key transport protocol, digital signature protocol, and key exchange protocol. Each of these protocols addresses specific security requirements such as encryption, authentication, and key sharing between two communicating parties, referred to as Alice and Bob.
The first protocol, the Basic Key Transport Protocol, ensures secure key exchange by using RSA for encrypting the session key and AES for encrypting the actual message. The second protocol, the RSA Signature Protocol, provides message integrity and authenticity through the use of RSA for signing and SHA256 as a hashing function. The third protocol, the Diffie-Hellman Key Exchange Protocol, enables Alice and Bob to establish a shared secret over an insecure channel, which can then be used for symmetric encryption using AES. Please notice that the implementation of both RSA and AES in all these 3 protocols has been taken from previous laboratories or exercise sessions.

2 RSA and AES

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm widely used for secure data transmission. It uses two keys: a public key for encryption and a private key for decryption. The keys are mathematically linked, but the private key cannot be easily derived from the public key, ensuring the security of the system. The security of RSA relies on the difficulty of factoring large numbers (specifically the product of two large primes), which makes it nearly impossible for an attacker to deduce the private key from the public key within a feasible timeframe. RSA 2048 is widely used because it offers strong security while balancing computational efficiency.
AES (Advanced Encryption Standard) is a symmetric encryption algorithm that encrypts data using a single key for both encryption and decryption. It is fast and highly secure, making it ideal for encrypting large amounts of data in real-time communication.

3 Basic Key Transport Protocol

The goal of the Basic Key Transport Protocol is to securely exchange a session key between two parties, Alice and Bob, using RSA for key transport and AES for encrypting the actual communication. In this protocol, the session key is a temporary, symmetric key used to encrypt the plaintext message, ensuring that the message remains confidential during transmission.

3.1 Protocol overview

By combining RSA for the secure transport of the session key and AES for efficient encryption of the message, this protocol achieves both security and performance. RSA protects the session key from being intercepted, while AES provides fast and secure encryption of the message itself. This hybrid approach leverages the strengths of both algorithms, ensuring secure communication between Alice and Bob.

3.2 Code Implementation

Bob and Alice Object Definitions: The protocol is implemented by defining two objects: Bob and Alice, each with distinct responsibilities. Bob generates an RSA key pair (public and private keys) and shares his public key with Alice. Alice generates a random symmetric key (AES 128-bit), encrypts it with Bob's RSA public key, and sends it back to Bob. Both parties then use this shared symmetric key to encrypt and decrypt messages using AES. Bob's Key Generation and Symmetric Key Decryption Bob is responsible for creating an RSA key pair, sharing his public key, and decrypting the symmetric key that Alice sends back.

```
class BOB:
    def __init__(self):
        self.pub_key = None
        self.private_key = None
        self.sym_key = None

    def generateKeyPair(self, key_name="BOB_KEY"):
        self.key_name = key_name
        makeKeyFiles(key_name, 2048)  # Generate a 2048-bit RSA key pair
        self.pub_key = readKeyFile(key_name + '_pubkey.txt')
        self.private_key = readKeyFile(key_name + '_privkey.txt')

    def receiveAndDecryptEncryptedKey(self, enc_key):
        keySize, n, d = self.private_key
        decryptedBlocks = decryptMessage(enc_key[1], enc_key[0], (n, d))
        self.sym_key = int(decryptedBlocks)
        print("Bob received and decrypted symmetric key:", hex(self.sym_key))

    def encryptMessage(self, message):
        print("Bob encrypts message: ", message)
        message = message.encode('utf-8')
        ciphertext = []
        for chunk in chunkstring(message, 16):
            plaintext = chunk.hex()
            plaintext = int(plaintext, 16)
            ciphertext.append(encrypt(plaintext, self.sym_key))
        return ciphertext
```

Alice's Symmetric Key Generation and Encryption Alice generates a symmetric key for AES encryption, encrypts it using Bob's public RSA key, and sends it back to Bob.

```
class ALICE:
    def __init__(self):
        self.bob_pub_key = None
        self.sym_key = None
        self.encrypted_key = None

    def receivePublicKey(self, pub_key):
        self.bob_pub_key = pub_key
        print("Alice received BOB public key:", pub_key)

    def generateSymKey(self):
        self.sym_key = int(os.urandom(16).hex(), 16)  # Generates a 128-bit AES key
        print("Alice generated symmetric key:", hex(self.sym_key))

    def encryptSymKey(self):
        message = self.sym_key
        message = str(message)
        keySize, n, e = self.bob_pub_key
        encryptedBlocks = encryptMessage(message, (n, e))
        self.encrypted_key = len(message), encryptedBlocks
        print("Alice encrypted symmetric key:", self.encrypted_key)

    def receiveAndDecryptMessage(self, enc_message):
        print("Alice receives encrypted message: ", enc_message)
        decrypted_aes_string = ""
        for block in enc_message:
            decrypted_aes = decrypt(block, self.sym_key)
            decrypted_aes_string += bytes.fromhex(format(decrypted_aes, 'x')).decode('utf-8'
    )
        print("Alice decrypts it: ", decrypted_aes_string)
        return decrypted_aes_string
```

Symmetric Key Generation: Alice creates a random 128-bit AES key. Symmetric Key Encryption: Alice encrypts the symmetric key using Bob's RSA public key and sends it back to Bob for decryption.
Message Decryption: Once Bob encrypts a message using AES, Alice decrypts it using the shared symmetric key.

Key Exchange and Message Flow Bob Generates and Sends Public Key. Bob generates his RSA key pair and shares the public key with Alice

```
b = BOB()
b.generateKeyPair()
```

Alice Receives Bob's Public Key and Encrypts Symmetric Key: Alice receives Bob's public key,

generates a 128-bit symmetric key, and encrypts it using RSA. The encrypted key is then sent back to Bob.

```
a = ALICE()
a.receivePublicKey(b.pub_key)
a.generateSymKey()
a.encryptSymKey()
b.receiveAndDecryptEncryptedKey(a.encrypted_key)
```

Message Encryption and Decryption: Bob encrypts a message using the symmetric AES key and sends it to Alice. Alice decrypts the message using the shared key.

```
secret_message = "Let's meet tomorrow at noon at the same spot."
ciphertext = b.encryptMessage(secret_message)
decrypted_message = a.receiveAndDecryptMessage(ciphertext)
```

Output Example Symmetric Key Exchange: Alice generated and encrypted the symmetric key:

```
Alice encrypted symmetric key: (39, [318187283615557...])
```

Bob decrypted the symmetric key:

```
Bob received and decrypted symmetric key: 0xb12947ddb0614591e32528c735315877
```

Bob encrypted the message:

```
Bob encrypts message:  Let's meet tomorrow at noon at the same spot.
```

Alice received and decrypted the message:

```
Alice decrypts it:  Let's meet tomorrow at noon at the same spot.
```

4 RSA Signature Protocol

The purpose of the RSA Signature Protocol is to verify the authenticity and integrity of a message by using RSA for digital signatures and SHA256 for hashing. In this protocol, the sender (Bob) signs the message's hash using his private RSA key to make sure that the message has not been altered and confirming the sender's identity. The receiver (Bob) can then verify the signature using Alice's public key to ensure that the message is genuine and unmodified.

4.1 Protocol overview

This protocol involves the following steps:

1. **Message Hashing**: Before signing, the original message is passed through a hashing algorithm to generate a fixed-size hash value (message digest) that uniquely represents the message.

2. **Signing**: The hash value is encrypted with the sender's private key to create the digital signature. Since only the sender has access to their private key, the signature confirms the sender's identity.

3. **Verification**: Upon receiving the message and the digital signature, the receiver decrypts the signature using the sender's public key. This reveals the hash of the original message. Simultaneously, the receiver generates a hash of the received message using the same hashing algorithm.

4. **Validation**: The receiver compares the decrypted hash with the newly generated hash. If they match, the message is considered authentic and unaltered; if they don't, the message might have been altered with or the signature is not valid.

4.2 Code Implementation

BOB Class: The BOB class is responsible for generating RSA key pairs, computing a message digest using SHA-256, and signing that digest. The key features of Bob class include:

Key Generation: The *generateKeyPair* method creates a public/private key pair and stores them.

Digest Calculation: The *computeDigest* method computes the SHA-256 digest of a given message.

Signing the Digest: The *signDigest* method encrypts the computed digest using Bob's private key to create a digital signature.

```
from hashlib import sha256

class BOB:
    def __init__(self):
        self.pub_key = None
        self.private_key = None
        self.key_name = None
        self.digest = None
        self.signature = None

    def generateKeyPair(self, key_name="BOB_KEY"):
        self.key_name = key_name
        try:
            makeKeyFiles(key_name, 2048)
        except SystemExit: # this means that the file already exists
            pass
        finally:
            self.pub_key = readKeyFile(key_name + '_pubkey.txt')
            self.private_key = readKeyFile(key_name + '_privkey.txt')

    def computeDigest(self, msg):
        self.digest = sha256(msg.encode('utf-8')).hexdigest()
        print("BOB computed digest:", self.digest)

    def signDigest(self):
        keySize, n, d = self.private_key
        length = len(self.digest)
```

```
        self.signature = length, encryptMessage(self.digest, (n, d))
        print("BOB signed digest:", self.signature)
```

ALICE Class: The ALICE class is responsible for receiving Bob's public key, the message, and the signature. It also verifies the signature. The key features of ALICE class include:

Receiving Public Key: The *receivePublicKey* method accepts Bob's public key.

Message and Signature Reception: Methods to receive the message and the signature from Bob.

Signature Verification: The *verifySignature* method computes the digest of the received message and compares it to the decrypted digest.

```
class ALICE:
    def __init__(self):
        self.bob_pub_key = None
        self.bob_digest = None
        self.message = None

    def receivePublicKey(self, pub_key):
        self.bob_pub_key = pub_key
        print("Alice received BOB public key:", pub_key)

    def receiveMessage(self, msg):
        print("Alice received message:", msg)
        self.message = msg

    def receiveSignature(self, signature):
        print("Alice received signature:", signature)
        keySize, n, e = self.bob_pub_key
        self.bob_digest = decryptMessage(signature[1], signature[0], (n, e))
        print("Alice decrypts digest:", self.bob_digest)

    def verifySignature(self):
        digest = sha256(self.message.encode('utf-8')).hexdigest()
        print("Alice computes digest:", digest)
        return digest == self.bob_digest
```

Usage Scenario

Key Generation: Bob generates his RSA key pair and shares his public key with Alice.

Message Signing: Bob reads a message from a file, computes its digest, and signs it.

Signature Verification: Alice receives the message and the signature from Bob. She verifies the signature to confirm the integrity of the message.

```
a = ALICE()
b = BOB()

# BOB generates his key pair and sends his public key to ALICE
```

```
b.generateKeyPair()
a.receivePublicKey(b.pub_key)

# BOB computes the signature for message x and sends it to ALICE, along with the message itself
f = open(filename, "r")
x = f.read()
f.close()
b.computeDigest(x)
b.signDigest()

a.receiveMessage(x)
a.receiveSignature(b.signature)

if a.verifySignature():
    print("Signature is valid!")
else:
    print("Signature is not valid!")

# Test with a different message
a.receiveMessage("I am Paul!")
a.receiveSignature(b.signature)

if a.verifySignature():
    print("Signature is valid!")
else:
    print("Signature is not valid!")
```

Example Output: When Alice verifies the correct message:

```
Alice received BOB public key: (2048, ...)
BOB computed digest: a76da427a23eb45b6be7ff63c06642dc1aae808a54375a944d1e601cebfd15e8
BOB signed digest: (64, [...])
Alice received message: Let's meet tomorrow at noon at the same spot.
Alice received signature: (64, [...])
Alice decrypts digest: a76da427a23eb45b6be7ff63c06642dc1aae808a54375a944d1e601cebfd15e8
Alice computes digest: a76da427a23eb45b6be7ff63c06642dc1aae808a54375a944d1e601cebfd15e8
Signature is valid!
```

When Alice verifies an altered message:

```
Alice received message: I am Paul!
Alice received signature: (64, [...])
Alice decrypts digest: a76da427a23eb45b6be7ff63c06642dc1aae808a54375a944d1e601cebfd15e8
Alice computes digest: 99ade47652ae8175d69283cb18422b4edf314d0c11d819b9598cac8a68387a93
Signature is not valid!
```

5 Diffie-Hellman Key Exchange Protocol

Diffie-Hellman key exchange's goal is to securely establish a channel to create and share a key for symmetric key algorithms. Once it is established, they can use it to derive a symmetric key for encrypting their communication using AES. This ensures both confidentiality and efficiency in subsequent exchanges. The Diffie-Hellman protocol enables the secure generation of this shared key even if a third party is listening on the channel.

5.1 Protocol overview

This protocol involves the following steps:

1. **Public Parameters**: Alice and Bob agree on two public parameters: A large prime number, p A generator (or base), g. These values are public and can be known to anyone, including potential eavesdroppers.

2. **Private Keys**: Alice selects a private key a, which she keeps secret. Bob selects a private key b, which he also keeps secret.

3. **Public Keys**: Alice calculates her public key using $A = g^a \mod p$. Bob calculates his public key using $B = g^b \mod p$ Alice sends her public key A to Bob, and Bob sends his public key B to Alice. These public keys can be shared over a public channel.

4. **Shared Secret**: Alice computes the shared secret using Bob's public key B and her private key a: $s = B^a \mod p$. Bob computes the shared secret using Alice's public key A and his private key b: $s = A^b \mod p$. Both Alice and Bob end up with the same shared secret s, because $g^{ab} \mod p$ is mathematically equivalent regardless of the order in which the operations are performed. This shared secret can now be used as a key for symmetric encryption.

An eavesdropper who intercepts the public values A, B, p, and g cannot compute the shared secret s without knowing either a (Alice's private key) or b (Bob's private key), due to the difficulty of solving the discrete logarithm problem.

5.2 Code Implementation

Constants Setup: The Diffie-Hellman protocol relies on a large prime number P and a base A (generator) for exponentiation. For this implementation:
P: A 2048-bit prime number.
A: The base for exponentiation was chosen as A = 5, because 5 is a generator of $\mathbb{Z}_P$.
BOB and ALICE Class Definitions: Alice and Bob are implemented as classes, with methods for key generation, public key computation, key exchange, and symmetric encryption using the derived common

secret.

BOB Class:

```
P=0x61d373e962dfa5f....412 # full number omitted for space
A = 5 # base for exponentiation
class BOB:
    def __init__(self):
      self.public_key = None
      self.private_key = None
      self.alice_public_key = None
      self.common_secret = None
      self.aes_key = None
    def generateKey(self):
      self.private_key = random.randint(1, P)
      print("Bob generates private key:", self.private_key)
    def computePublicKey(self):
      self.public_key = pow(A, self.private_key, P)
      print("Bob computes public key:", self.public_key)
    def receiveAlicePublicKey(self, key):
      self.alice_public_key = key
    def computeCommonSecret(self):
      self.common_secret = pow(self.alice_public_key,self.private_key,P)
      print("Bob computes common secret:", self.common_secret)
    def deriveKey(self):
      self.aes_key = int(hashlib.md5(str(self.common_secret).encode()).hexdigest(), 16)
      print("Bob derived secret key:", hex(self.aes_key))
    def encryptMessage(self,message):
      print("Bob encrypts message: ", message)
      message = message.encode('utf-8')
      ciphertext = []
      for chunk in chunkstring(message, 16):
        plaintext = chunk.hex()
        plaintext = int(plaintext,16)
        ciphertext.append(encrypt(plaintext, self.aes_key))
      return ciphertext
```

ALICE Class:

```
class ALICE:
    def __init__(self):
      self.public_key = None
      self.private_key = None
      self.bob_public_key = None
      self.common_secret = None
      self.aes_key = None
    def generateKey(self):
      self.private_key = random.randint(1, P)
```

```
      print("Alice generates private key:", self.private_key)
    def computePublicKey(self):
      print(A, self.private_key, P, pow(A, self.private_key, P))
      self.public_key = pow(A, self.private_key, P)
      print("Alice computes public key:", self.public_key)
    def receiveBobPublicKey(self, key):
      self.bob_public_key = key
    def computeCommonSecret(self):
      self.common_secret = pow(self.bob_public_key,self.private_key,P)
      print("Alice computes common secret:", self.common_secret)
    def deriveKey(self):
      self.aes_key = int(hashlib.md5(str(self.common_secret).encode()).hexdigest(), 16)
      print("Alice derived secret key:", hex(self.aes_key))
    def receiveAndDecryptMessage(self, enc_message):
      print("Alice receives encrypted message: ", enc_message)
      decrypted_aes_string = ""
      for block in enc_message:
        decrypted_aes = decrypt(block, self.aes_key)
        decrypted_aes_string += bytes.fromhex(format(decrypted_aes,'x')).decode('utf-8')
      print("Alice decrypts it: ",decrypted_aes_string)
      return decrypted_aes_string
```

Key Exchange: Alice receives Bob's public key, and Bob receives Alice's public key. Both Alice and Bob compute a common secret using the other party's public key and their own private key. The common secret is hashed using MD5 to derive a 128-bit AES key, which will be used for symmetric encryption.

Message Encryption/Decryption: Bob encrypts a message using the derived AES key, converting the message into hexadecimal format and performing XOR encryption with the AES key. Alice decrypts the message using the same AES key, reversing the XOR operation to retrieve the original message.

```
a = ALICE()
b = BOB()

# Key generation
a.generateKey()
b.generateKey()

# Public key computation
a.computePublicKey()
b.computePublicKey()

# Key exchange
a.receiveBobPublicKey(b.public_key)
b.receiveAlicePublicKey(a.public_key)

# Key agreement
```

```
a.computeCommonSecret()
b.computeCommonSecret()

# Key derivation
a.deriveKey()
b.deriveKey()

# Message exchange (encryption and decryption)
f = open(filename, "r")
secret_message = f.read()
f.close()

# Bob encrypts and Alice decrypts the message
ciphertext = b.encryptMessage(secret_message)
decrypted_message = a.receiveAndDecryptMessage(ciphertext)
```

6 Conclusions

In this lab, we successfully implemented three critical cryptographic protocols: the Basic Key Transport Protocol, the RSA Signature Protocol, and the Diffie-Hellman Key Exchange Protocol.

Basic Key Transport Protocol: We utilized RSA for securely exchanging a session key and AES for efficient message encryption. This hybrid approach ensured that even if an adversary intercepted the key exchange, the confidentiality and integrity of the session key and subsequent communications remained intact. Bob and Alice effectively exchanged a symmetric key through RSA encryption, demonstrating a robust mechanism for secure communication.

RSA Signature Protocol: This protocol allowed for the verification of message authenticity and integrity. By employing RSA for signing and SHA-256 for hashing, we established a reliable method for Bob to sign messages and for Alice to verify their authenticity. The implementation highlighted the importance of digital signatures in providing non-repudiation and ensuring that messages have not been altered in transit.

Diffie-Hellman Key Exchange Protocol: This protocol enabled Alice and Bob to establish a shared secret over an insecure channel without transmitting the secret itself. The successful key exchange facilitated the derivation of a symmetric AES key, which was then used for secure communication. The implementation demonstrated the practical application of the Diffie-Hellman protocol in real-world scenarios, ensuring confidentiality despite potential eavesdroppers.