Kyle Jennings

Zarek Lazowski

Evan Morris

Public Ciphers

For this assignment, we created programs that simulate the Diffie-Hellman Key Exchange, RSA encryption, and various vulnerabilities for both of these protocols. Task1() shows how a simple DH Key Exchange operates, without any attackers. Task2_1() and Task2_2() shows two different ways of how a machine in the middle attack would render both Alice's and Bob's data vulnerable. Task3_1() shows that our homebrew RSA implementation works. Task3_2() shows how a machine in the middle attack would work with RSA encryption, by having Mallory encrypt and send her own s value to Alice.

1. For task 1, how hard would it be for an adversary to solve the Diffie Hellman Problem (DHP) given these parameters? What strategy might the adversary take?
    - It shouldn't be too hard as you know A, B, p, and g. All you would really need to do is solve for Alice and Bob's a and b. This would require using the Extended Euclidean algorithm to solve for $x = g^b$. It gets harder to crack with larger numbers, however once you find a or b, it should be simple to solve for s the same way Alice and Bob had.

2. For task 1, would the same strategy used for the tiny parameters work for the p and g? Why or why not?
    - It could, but not nearly as easily as with smaller numbers. Much larger numbers would require much more computation to reverse engineer.

3. For task 2, why were these attacks possible? What is necessary to prevent it?
    - These tasks are possible because the MITM (Mallory) is able to snoop on the traffic between Alice and Bob. The data they are using to generate a "secret" key to communicate through is in plaintext. As long as Mallory knows what protocol they are attempting to use, she can intervene and pose as both Alice and Bob.
    - In order to prevent this from happening, Alice and Bob would need to share the necessary information using another secret key that they ideally only them and nobody else knows.

4. For task 3 part 1, while it's very common for many people to use the same value for e in their key (common values are 3, 7, 216+1), it is very bad if two people use the same RSA modulus n. Briefly describe why this is, and what the ramifications are.
    - Assuming e and d are both prime, it is possible to find s and t that together undoes the cipher text. If we choose s and t such that es + dt = 1, then we would be able to combine cipher texts to decrypt the original message. This could look like C_1^s and C_2^t which is equivalent to M^(es) and M^(dt). By combining these terms together like M^(es) * M^(dt), we can find M^1 % n. However, it should be noted that the attacker would need to know e, d, and n.

5. Give another example of how RSA's malleability could be used to exploit a system (e.g. to cause confusion, disruption, or violate integrity).
    - Besides allowing an attacker to snoop on conversations between two parties, RSAs malleability allows attackers to encrypt their own message to cause a misunderstanding between the two parties and encrypt or send garbage to cause the two parties to not be able to communicate anymore.

6. Malleability can also affect signature schemes based on RSA. Consider the scheme: $Sign(m, d) = m^d \% n$. Suppose Mallory sees the signatures for two messages $m1\ and\ m2$. Show how Mallory can create a valid signature for a third message, $m3$ =$m1 \cdot m2$.
    - You can define some third message $m_3 = m_1 * m_2$
    - Mallory can see both signatures $s_1 = {m_1}^d \ \% \ n$ and $s_2 = {m_2}^d \ \% \ n$
    - To find the signature of m_3:

$$ {m_3}^d \ \% \ n $$

$$ (m_1 * m_2)^d \ \% \ n $$

$$ \left({m_1}^d * {m_2}^d\right) \% \ n $$

$$ \left({m_1}^d \ \% \ n\right) * \left({m_2}^d \ \% \ n\right) \% \ n $$

$$ (s_1 * \ s_2) \ \% \ n $$

    - Here we see an operation that we can perform on the signatures in order to get a valid signature for a known message: $m_3 = \ m_1 * \ m_2$. So $s_3 = \ (s_1 * \ s_2) \ \% \ n$.

Code

```
import random
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import sys

from large_primes import get_large_prime


class User:
    """ Class for storing relevant client information in the exchange
    """
    def __init__(self, p=35, g=5):
        # save the p and g values
        self.p = p
        self.g = g

        # get the seed for the values to be exchanged
        self.c = random.randint(1, 1000)

    def get_C(self):
        # calculate the value to be eventually exchanged
        return (self.g ** self.c) % self.p

    def compute_s(self, C: int):
        # get the unhashed value for the key
        self.s = (C ** self.c) % self.p

    def calc_key(self):
        # hash the initial key value
        digest = hashes.Hash(hashes.SHA256())
        digest.update(self.s.to_bytes(128, 'big'))
        self.key = digest.finalize()[:16]

    def send_text(self, plaintext: str):
        # encrypt the text `m` with the key we generated earlier
        encryptor = Cipher(algorithms.AES(self.key), modes.CBC(self.key)).encryptor()
        return encryptor.update(plaintext.encode("UTF-8")) + encryptor.finalize()

    def receive_text(self, ciphertext: bytes):
        # decrypt the the cipher text and print it
        decryptor = Cipher(algorithms.AES(self.key), modes.CBC(self.key)).decryptor()
        print((decryptor.update(ciphertext) + decryptor.finalize()).decode())

class RSA:
    public_key: int
    private_key: int

    def __init__(self, key_size=512):
        self.E = 65537
        self.key_gen(key_size)

    def key_gen(self, key_size):
        # Lets pretend I can generate large primes
        P = get_large_prime(key_size)
        Q = get_large_prime(key_size)
```

```python
        # Generate a public key
        self.public_key_gen(P, Q)

        # Generate a private key
        self.private_key_gen(P, Q)

    def public_key_gen(self, P, Q):
        # Public keys are two large prime numbers multiplied by eachother
        self.public_key = P * Q

    def private_key_gen(self, P, Q):
        # E*D mod(((P-1) * (Q-1)) = 1
        # Solve for D, the private key: D = (1/E) invmod ((P-1) * (Q-1))
        self.private_key = pow(self.E, -1, (P-1) * (Q-1))

    def pub_encrypt(self, m: str) -> bytes:
        P = int.from_bytes(m.encode("utf8"), "little")
        C = pow(P, self.E, self.public_key)
        return C.to_bytes(sys.getsizeof(C), "little")

    def priv_decrypt(self, c: bytes) -> str:
        C = int.from_bytes(c, "little")
        P: int = pow(C, self.private_key, self.public_key)
        return P.to_bytes(len(c), "little").decode()


def task1():
    # assign the two large p and g values
    p = int('B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA 52D23B61
6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C D49B83BF ACCBDD7D
90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F F55BCCC0 A151AF5F 0DC8B4BD
45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC 2E4A4371'.replace(" ", ""), 16)
    g = int('A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF 14266D31
266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F 9E2749F4 D7FBD7D3
B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01 69B6A28A D662A4D1 8E73AFA3
2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB 22B3B2E5'.replace(" ", ""), 16)

    # create Alice and Bob with the p and g values
    Alice = User(p, g)
    Bob = User(p, g)

    # Get the exchange values
    A = Alice.get_C()
    B = Bob.get_C()

    # calculate the key primitive
    Alice.compute_s(B)
    Bob.compute_s(A)

    # hash the key
    Alice.calc_key()
    Bob.calc_key()

    # make sure the keys are the same
    assert Alice.key == Bob.key
```

```python
    # try to exchange messages
    Bob.receive_text(Alice.send_text('Hi Bob!AAAAAAAAA'))
    Alice.receive_text(Bob.send_text('Hi Alice!AAAAAAA'))

def task2_1():
    '''
    Mallory changes A/B: key becomes SHA-256(0), and you can decrypt both messages.
    '''

    p = int('B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA 52D23B61
6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C D49B83BF ACCBDD7D
90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F F55BCCC0 A151AF5F 0DC8B4BD
45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC 2E4A4371'.replace(" ", ""), 16)
    g = int('A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF 14266D31
266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F 9E2749F4 D7FBD7D3
B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01 69B6A28A D662A4D1 8E73AFA3
2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB 22B3B2E5'.replace(" ", ""), 16)

    Alice = User(p, g)
    Bob = User(p, g)

    A = Alice.get_C()
    B = Bob.get_C()

    # machine in middle
    A = p
    B = p

    Alice.compute_s(B)
    Bob.compute_s(A)

    Alice.calc_key()
    Bob.calc_key()

    assert Alice.key == Bob.key

    Bob.receive_text(Alice.send_text('Hi Bob!AAAAAAAAA'))
    Alice.receive_text(Bob.send_text('Hi Alice!AAAAAAA'))

def task2_2():
    '''
    Mallory sends a new p to Bob, and intercepts messages from both.
    '''

    p = int('B10B8F96 A080E01D DE92DE5E AE5D54EC 52C99FBC FB06A3C6 9A6A9DCA 52D23B61
6073E286 75A23D18 9838EF1E 2EE652C0 13ECB4AE A9061123 24975C3C D49B83BF ACCBDD7D
90C4BD70 98488E9C 219A7372 4EFFD6FA E5644738 FAA31A4F F55BCCC0 A151AF5F 0DC8B4BD
45BF37DF 365C1A65 E68CFDA7 6D4DA708 DF1FB2BC 2E4A4371'.replace(" ", ""), 16)
    g = int('A4D1CBD5 C3FD3412 6765A442 EFB99905 F8104DD2 58AC507F D6406CFF 14266D31
266FEA1E 5C41564B 777E690F 5504F213 160217B4 B01B886A 5E91547F 9E2749F4 D7FBD7D3
B9A92EE1 909D0D22 63F80A76 A6A24C08 7A091F53 1DBF0A01 69B6A28A D662A4D1 8E73AFA3
2D779D59 18D08BC8 858F4DCE F97C2A24 855E6EEB 22B3B2E5'.replace(" ", ""), 16)

    # Mallory sends a tampered g value to Bob
    g_b = 1

    Alice = User(p, g)
```

```python
    Bob = User(p, g_b)

    # Mallory intercepts the message to Bob and creates two sessions, one to Alice and
the other to Bob
    Mal_A = User(p, g)
    Mal_B = User(p, g_b)

    A = Alice.get_C()
    B = Bob.get_C()
    A_m = Mal_A.get_C()
    B_m = Mal_B.get_C()

    # Alice and Mallory have their own unique s value
    Alice.compute_s(A_m)
    Mal_A.compute_s(A)

    # Bob and Mallory have their own unique s value
    Bob.compute_s(B_m)
    Mal_B.compute_s(B)

    # Everyone calculates their own keys
    Alice.calc_key()
    Mal_A.calc_key()

    Bob.calc_key()
    Mal_B.calc_key()

    assert Alice.key == Mal_A.key
    assert Bob.key == Mal_B.key

    # Mallory can receive both Alice's and Bob's messages
    print('Mallory sees: ', end='')
    Mal_A.receive_text(Alice.send_text('Hi Bob!AAAAAAAAA'))

    print('Mallory sees: ', end='')
    Mal_B.receive_text(Bob.send_text('Hi Alice!AAAAAAA'))


def task3_1():
    Alice = RSA()
    Bob = RSA()

    # encrypt with bob's public key, then decrypt with bob's private key
    ciphertext = Bob.pub_encrypt("Hello, world!")
    print(Bob.priv_decrypt(ciphertext))

    # encrypt with alice's public key, then decrypt with alice's private key
    ciphertext = Alice.pub_encrypt("Goodbye, world!")
    print(Alice.priv_decrypt(ciphertext))

def task3_2():
    Alice = User()
    Mallory = User()

    # Alice sends her public key to Bob (and Mallory). Sends n and e.
    Alice_rsa = RSA()
```

```python
    # Bob selects a random number s, and encrypts it with Alice's public key
    s_b = str(random.randint(0, 1000))
    c_b = Alice_rsa.pub_encrypt(s_b)

    # Mallory intercepts the message, encrypts her own s, and sends it to Alice
    s_m = str(2)
    Mallory.s = int(s_m)
    c_m = Alice_rsa.pub_encrypt(s_m)

    # Alice decrypts Mallory's message to get s, and uses that to select the key
    Alice.s = int(Alice_rsa.priv_decrypt(c_m)[0])

    # Both Alice and Mallory calculate their keys
    Alice.calc_key()
    Mallory.calc_key()

    # Mallory decrypts the message intended to be sent to Bob
    Mallory.receive_text(Alice.send_text('Hi Bob!AAAAAAAAA'))


if __name__ == '__main__':
    task3_2()
```