Introduction to Computer Security

Lab 2 - Exploring Public Key Cryptography

Report
of
Andreas Wilhelm

February 5, 2019

You can find the code and everything else also on GitHub under the link *https://github.com/awilsee/CSec*. Maybe more convenient for you.

1 Implement Diffie Hellman Key Exchange

```
import hashlib
import os

from Crypto.Cipher import AES


aes_bytes = 16
#p = 37
#g = 5
p = int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D1
", 16)
g = int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564
", 16)


def add_pkcs7(input):
    num_bytes = aes_bytes - (len(input) % aes_bytes)
    fil_bytes = bytes()
    for x in range(num_bytes):
        fil_bytes += bytes([num_bytes])
    input += fil_bytes
    return input


def remove_pkcs7(input):
    pad_byte = input[len(input) - 1]
    return input[:len(input) - pad_byte]


if __name__ == '__main__':
    privA = 6
    privB = 15

    shared_a = (g ** privA) % p
    shared_b = (g ** privB) % p

    calc_shared_secA = (shared_b ** privA) % p
    calc_shared_secB = (shared_a ** privB) % p

    hash_a = hashlib.sha256("{}".format(int(calc_shared_secA)).encode()).digest()
    hash_b = hashlib.sha256("{}".format(int(calc_shared_secB)).encode()).digest()

    print(hash_a)
    print(hash_b)

    m_a = b"Hi Bob!"
    m_b = b"Hi Alice!"
```

```

    m_a = add_pkcs7(m_a)
    m_b = add_pkcs7(m_b)

    #encrypting
    init_iv = os.urandom(16)
    cipher_a = AES.new(hash_a[:16], AES.MODE_CBC, init_iv)
    crypt_a = cipher_a.encrypt(m_a)

    cipher_b = AES.new(hash_b[:16], AES.MODE_CBC, init_iv)
    crypt_b = cipher_b.encrypt(m_b)

    #decrypting
    cipher_a = AES.new(hash_a[:16], AES.MODE_CBC, init_iv)
    msg_a = cipher_a.decrypt(crypt_b)

    cipher_b = AES.new(hash_b[:16], AES.MODE_CBC, init_iv)
    msg_b = cipher_b.decrypt(crypt_a)

    print(remove_pkcs7(msg_a))
    print(remove_pkcs7(msg_b))
```

Listing 1: Code of 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?
With p = 37 and g = 5 it could be relatively easy to guess respectively to calculate these numbers. Because these two numbers have to be primes and the adversary also knows the algortihm behind it, so he can try out some numbers. Because they are very small with only some bits it's possible in a resonable time.

Would the same strategy used for the tiny parameters work here? Why or why not?
No, it wouldn't because the prime numbers are now to large, so there are two many possibilities to calculate, it would take several years to get the right numbers.

2 Implement MITM Key Fixing & Negotiated Groups

```
import hashlib
import os

from Crypto.Cipher import AES


aes_bytes = 16
#p = 37
#g = 5
p = int("
    B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D1
    ", 16)
g = int("
    A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564
    ", 16)

#private keys of Alice and Bob
privA = 6
privB = 15
```

```


def add_pkcs7(input):
    num_bytes = aes_bytes - (len(input) % aes_bytes)
    fil_bytes = bytes()
    for x in range(num_bytes):
        fil_bytes += bytes([num_bytes])
    input += fil_bytes
    return input


def remove_pkcs7(input):
    pad_byte = input[len(input) - 1]
    return input[:len(input) - pad_byte]


def enc_send_dec(hash_a, hash_b, hash_m):
    m_a = add_pkcs7(b"Hi Bob!")
    m_b = add_pkcs7(b"Hi Alice!")

    #encrypting and sending..
    init_iv = os.urandom(16)
    cipher_a = AES.new(hash_a[:16], AES.MODE_CBC, init_iv)
    crypt_a = cipher_a.encrypt(m_a)

    cipher_b = AES.new(hash_b[:16], AES.MODE_CBC, init_iv)
    crypt_b = cipher_b.encrypt(m_b)

    #Mallory decrypt messages
    print("Mallory decrypts msgs")
    cipher_m = AES.new(hash_m[:16], AES.MODE_CBC, init_iv)
    msg = cipher_m.decrypt(crypt_a)
    print(remove_pkcs7(msg))
    cipher_m = AES.new(hash_m[:16], AES.MODE_CBC, init_iv)
    msg = cipher_m.decrypt(crypt_b)
    print(remove_pkcs7(msg))

    #decrypting
    cipher_a = AES.new(hash_a[:16], AES.MODE_CBC, init_iv)
    msg_a = cipher_a.decrypt(crypt_b)

    cipher_b = AES.new(hash_b[:16], AES.MODE_CBC, init_iv)
    msg_b = cipher_b.decrypt(crypt_a)

    print("Alice and Bob decrypts msg")
    print(remove_pkcs7(msg_a))
    print(remove_pkcs7(msg_b))


def compute_keys(g_local, key_fixing):
    # A send shared
    shared_a = (g_local ** privA) % p
    # B send shared
    shared_b = (g_local ** privB) % p

    if key_fixing:
        # Mallory modifies A and B to p
        shared_a = p
```

```
        shared_b = p

    # computes their shared secret
    calc_shared_secA = (shared_b ** privA) % p
    calc_shared_secB = (shared_a ** privB) % p

    print(shared_a)
    print(calc_shared_secA)

    hash_a = hashlib.sha256("{}".format(int(calc_shared_secA)).encode()).digest()
    hash_b = hashlib.sha256("{}".format(int(calc_shared_secB)).encode()).digest()

    print(hash_a)
    print(hash_b)

    # creating key for Mallory
    if key_fixing:
        g_local = p
    calc_shared_secM = (g_local ** 10) % p  # == 0
    hash_m = hashlib.sha256("{}".format(int(calc_shared_secM)).encode()).digest()
    return hash_a, hash_b, hash_m


if __name__ == '__main__':
    print("Part A")
    hash_a, hash_b, hash_m = compute_keys(g, True)
    enc_send_dec(hash_a, hash_b, hash_m)

    g_local = 1
    print("\nPart B g={}".format(g_local))
    hash_a, hash_b, hash_m = compute_keys(g_local, False)
    enc_send_dec(hash_a, hash_b, hash_m)

    g_local = p
    print("\nPart B g={}".format(g_local))
    hash_a, hash_b, hash_m = compute_keys(g_local, False)
    enc_send_dec(hash_a, hash_b, hash_m)

    g_local = p - 1
    print("\nPart B g={}".format(g_local))
    hash_a, hash_b, hash_m = compute_keys(g_local, False)
    enc_send_dec(hash_a, hash_b, hash_m)
```

Listing 2: Code of task 2

Why were these attacks possible? What is necessary to prevent them?
All these attacks works in the same way, because with to calculate the modulo of a multiple of the same number results in 0 or in the other examples 1. So the private secret key doesn't really come into effect.
One way to prevent this would be a check if g is a p, 0 or 1. or the results afterwards.

3 Implement textbook RSA & MITM Key Fixing via Malleability

```python
# http://inventwithpython.com/hacking (BSD Licensed)
import hashlib
import os

import PrimesRabinMiller as prm
from Crypto.Cipher import AES

from task2 import add_pkcs7
from task2 import remove_pkcs7

e = 65537
KEY_SIZE = 1024 # 128 bytes
BYTE_SIZE = 256 # One byte has 256 different values.

def gcd(a, b):
    # Return the GCD of a and b using Euclid's Algorithm
    while a != 0:
        a, b = b % a, a
    return b


def findModInverse(a, m):
    # Returns the modular inverse of a % m, which is
    # the number x such that a*x % m = 1

    if gcd(a, m) != 1:
        return None # no mod inverse if a & m aren't relatively prime

    # Calculate using the Extended Euclidean Algorithm:
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m
    while v3 != 0:
        q = u3 // v3 # // is the integer division operator
        v1, v2, v3, u1, u2, u3 = (u1 - q * v1), (u2 - q * v2), (u3 - q * v3), v1, v2, v3
    return u1 % m


def gen_prime_key(key_size):
    #Creating two prime numbers, p and q. Calculate n = p * q.
    print('Generating p prime...')
    p = prm.generate_prime_number(key_size)
    print('Generating q prime...')
    q = prm.generate_prime_number(key_size)
    n = p * q

    #Calculate d, the mod inverse of e, e is static
    print('Calculating d that is mod inverse of e...')
    d = findModInverse(e, (p - 1) * (q - 1))
    public_k = (n, e)
    private_k = (n, d)
```

```
    print('Public key :', public_k)
    print('Private key:', private_k)

    return public_k, private_k


def get_blocks_from_string(msg, blockSize=KEY_SIZE // 8):
    # Converts a string message to a list of block integers. Each integer
    # represents 128 (or whatever blockSize is set to) string characters.
    msgBytes = msg.encode('ascii') # convert the string to bytes
    blockInts = []
    for blockStart in range(0, len(msgBytes), blockSize):
        # Calculate the block integer for this block of text
        blockInt = 0
        for i in range(blockStart, min(blockStart + blockSize, len(msgBytes))):
            blockInt += msgBytes[i] * (BYTE_SIZE ** (i % blockSize))
        blockInts.append(blockInt)
    return blockInts


def get_string_from_blocks(blockInts, msgLen, blockSize=KEY_SIZE // 8):
    # Converts a list of block integers to the original message string.
    # The original message length is needed to properly convert the last
    # block integer.
    message = []
    for blockInt in blockInts:
        blockMessage = []
        for i in range(blockSize - 1, -1, -1):
            if len(message) + i < msgLen:
                # Decode the message string for the 128 (or whatever
                # blockSize is set to) characters from this block integer.
                asciiNumber = blockInt // (BYTE_SIZE ** i)
                blockInt = blockInt % (BYTE_SIZE ** i)
                blockMessage.insert(0, chr(asciiNumber))
        message.extend(blockMessage)
    return ''.join(message)


def encrypt_msg(msg, pub_key):
    encrypted_blocks = []

    for blocks in get_blocks_from_string(msg):
        # c = plain ^ e mod n
        encrypted_blocks.append(pow(blocks, pub_key[1], pub_key[0]))
    return encrypted_blocks


def decrypt_msg(block_msg, priv_key, msgLen):
    decrypt_blocks = []
    for blocks in block_msg:
        # plain = c ^ d mod n
```

```
            decrypt_blocks.append(pow(blocks, priv_key[1], priv_key[0]))
    return get_string_from_blocks(decrypt_blocks, msgLen)


if __name__ == '__main__':

    public_k, private_k = gen_prime_key(KEY_SIZE) #n,e n,d

    msg1 = "Hello World!"
    msg2 = "This is a string which includes more than 128 Bytes. It is used to test if the block building also works if you have more than block_size Bytes."

    enc_msg = encrypt_msg(msg2, public_k)

    dec_msg = decrypt_msg(enc_msg, private_k, len(msg2))
    print(dec_msg)

    print("\nPartB")
    privS = 21
    shared_c = pow(privS, public_k[1], public_k[0])

    #mallory modifies c
    shared_c = public_k[0]

    s_a = pow(shared_c, private_k[1], private_k[0])
    s_m = 0
    hash_a = hashlib.sha256("{}".format(int(s_a)).encode()).digest()
    hash_m = hashlib.sha256("0".encode()).digest()

    #encrypting, sending and decrypting with key of mallory
    init_iv = os.urandom(16)
    cipher = AES.new(hash_a[:16], AES.MODE_CBC, init_iv)
    crypt = cipher.encrypt(add_pkcs7(bytes(msg2, 'utf-8')))

    cipher = AES.new(hash_m[:16], AES.MODE_CBC, init_iv)
    dec = cipher.decrypt(crypt)

    print(remove_pkcs7(dec).decode('utf-8'))
```

Listing 3: Code of task 3

```
from random import randrange, getrandbits


def is_prime(n, k=128):
    """ Test if a number is prime
        Args:
            n -- int -- the number to test
            k -- int -- the number of tests to do
        return True if n is prime
    """
    # Test if n is not even.
    # But care, 2 is prime !
    if n == 2 or n == 3:
        return True
    if n <= 1 or n % 2 == 0:
        return False
    # find r and s
```

```python
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2
    # do k tests
    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while j < s and x != n - 1:
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True


def generate_prime_candidate(length):
    """ Generate an odd integer randomly
        Args:
            length -- int -- the length of the number to generate, in bits
        return a integer
    """
    # generate random bits
    p = getrandbits(length)
    # apply a mask to set MSB and LSB to 1
    p |= (1 << length - 1) | 1
    return p


def generate_prime_number(length=1024):
    """ Generate a prime
        Args:
            length -- int -- length of the prime to generate, in bits
        return a prime
    """
    p = 4
    # keep generating while the primality test fail
    while not is_prime(p, 128):
        p = generate_prime_candidate(length)
    return p
```

Listing 4: Code of task 3 - PrimesRabinMiller

3.1 A

While it's very common for many people to share an e (common values are 3,7, 2^{16}+1), it is very bad if two people share an RSA modulus n. Briefly describe why this is, and what the ramifications are.

It is common for e because it simplifies the encryption and doesn't take too long for encryption for example if you use 2^{16}+1 there are only two bits 1. n shouldn't be shared because then you take the same both prime numbers. If you have different msg from different people with the same key you can start guessing and get the key more easily.

3.2 B

Give another example of how RSA's malleability could be used to exploit a system (e.g. to cause confusion, disruption, or violate integrity).
In the listing 3 starting at line 118 the example of malleability is that Mallory change c' to n which results in 0 for s as explained earlier.
Another example would be if you think on an auction where anybody send their encrypted bits. Mallory can multiply the bid from someone else even without encryption and knowing how much the other bid.

*Suppose Mallory sees the signatures for two message m1 and m2. Show how Mallory can create a valid signature for a third message, m3 = m1 * m2.*
If the message is signed, Mallory can use the same attack to get to the messages as well as write some messages.

3.3 C

Briefly justify whether either of the following key exchange protocols do or do not provide forward secrecy.
Well, neither protocol prevents brute-force attacks on the underlying ciphers, however with such a session key you prevent that your communications from the past can't be read by compromising your private key. It is mostly combined with the Diffie Hellman or even better with Elliptic Curve Diffie Hellman.
If an attacker can get the private key of the server the RSA isn't secure anymore because the session key correspond with the keys and therefore all sessions can be decrypted.
However you can use Perfect Forward Secrecy in this case which removes the link between servers and sessions key.

4 Performance of RSA and AES

On figure 1 you can see that the larger the keys sizes the slower it gets. Whereas the throughout gets higher the larger the block size is.
On figure 2 you can see that throughput is decreasing the larger the key size gets. Additionally the verification operation is significantly faster than signing.
Though it's hard to compare them in detail, because the speed for RSA is measured in operations whereas the speed for AES is measured in kbit/s so its two different methods. Furthermore block size and key size is also not the same. Conspicuous is that at AES the throughput is increasing with the larger block size and at RSA it's decreasing.

Figure 1: AES

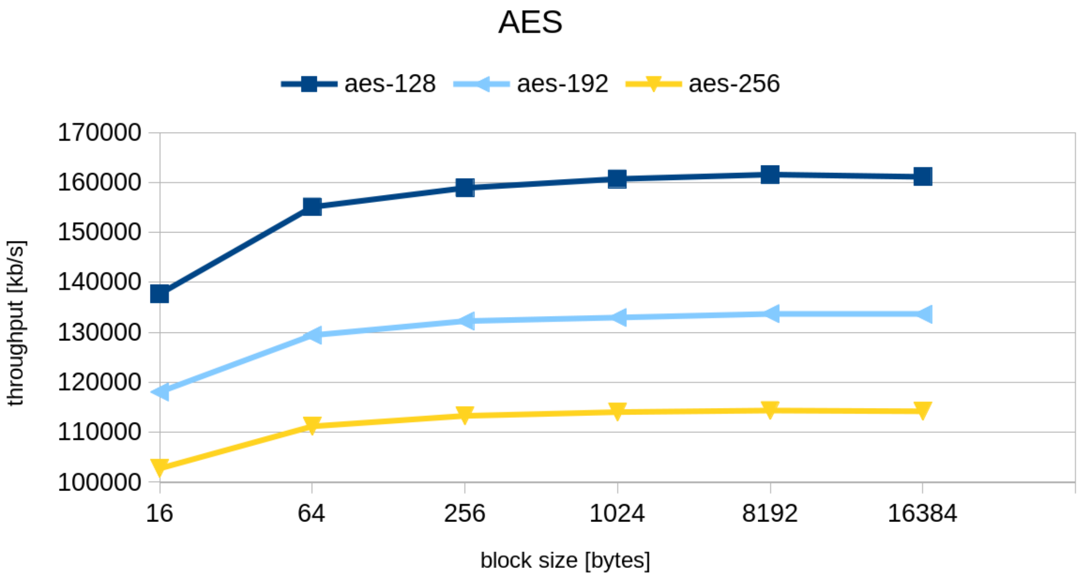

Figure 2: RSA