Applied Information Security and Cryptography

Lab 03: Public-key encryption

In this lab, you will experiment different solutions for achieving confidentiality on a communication channel through public-key encryption. The main advantage of public-key encryption is that you don't have to share a secret key in advance: you just publish your public key and everybody can send confidential messages to you. Of course, the other party should be convinced that the key is indeed "your" public-key and not "someone else" public-key. This is another problem that we will tackle in next labs.

You will need the Python "cryptography" module to perform AES encryption and decryption. This can be installed with: pip install cryptography (or pip3 install cryptography according to your configuration). See instructions here: https://cryptography.io/en/latest/

RSA encryption

Rivest-Shamir-Adleman (RSA) is probably the simplest algorithm for performing public-key encryption. Once you have a public key (N, e), where N and e are just "big" integer numbers (N must be big, e usually is not so big), you have to map your message to one (or more) "big" integer m such that $m < N$ and the ciphertext is simply:

$$c = m^e \bmod N$$

In Python this is very convenient, since the language manages arbitrary length integer numbers and the built-in function:

```
pow(base, exp, mod)
```

can be used to compute: `base**exp % mod` very efficiently (remember the modular exponentiation algorithm). At the start of the lab, the RSA public key of the instructor will be published. As a warm-up, encrypt a message with this public key and send it to the instructor. You can post the encrypted message on the public chat (just copy the resulting big number, pay attention to include all digits), only the instructor will be able to decrypt it.

Since with RSA one of the most time consuming and error prone tasks can be the encoding (and subsequent decoding) of the message as a big integer number, the Python script AISC_03.py provides two functions for it:

```
m = encodeText(s, bitlen)
```

Encode the string s into a list of integers each representable with bitlen-8 bits. Bitlen should be the length in bits of the RSA modulo N (e.g., 1024 bits). For the initial tests, it is suggested to limit the length of the message string s to bitlen/8 – 1 characters, so that it can be encoded in a single integer.

```
s = decodeText(m, bitlen)
```

Decode the list of integers m into the string s. It is guaranteed that decodeText(encodeText(s, bitlen), bitlen) == s.

After sending your first RSA encrypted message, you need to publish your public key to receive an encrypted answer. This requires generating your key pair, which is probably the most complex part of RSA.

First, you need generating two large prime numbers p and q with the right size. If you want an RSA key of n bits, p and q should be $n/2$ bits each. Moreover, it is required that both p and q are larger than $\sqrt{2} \cdot 2^{n/2-1}$, so that their product will be larger than 2^{n-1} (i.e., a n-bit number with the MSB set to one). Lastly, you should generate uniformly distributed and unpredictable numbers. The usual Python random number generator is not sufficiently unpredictable for cryptographic applications. Luckily, most operating systems implement a secure random number generator that exploits some truly unpredictable hardware sources. In Python, this is accessible in a cross-platform way using os.urandom (So, even if you are on Windows, you don't have to do a weird function call to a Windows library).

The Python script AISC_03.py provides a function for generating an unpredictable odd integer of the required size:

`generate_prime_candidate(length)`

If you need an RSA key of n bits, you should use length = $n/2$.

At this point, you need to test if your candidate is indeed a prime number. An efficient way is to use the Miller-Rabin test. Here is a pseudo-code (this is slightly different from the version on the slides, but you can check that the test is equivalent):

```
Input(n, k)
    Find s,r such that (n − 1) = 2^s ∗ r
    Repeat k times
        a = random number in (2, n − 1)
        x = a^r mod n
        # if a^r = +−1 mod n then x may be prime so test next a
        if x != 1 and x != n − 1:
            j = 1
             # if a^2jr = −1 mod n then x may be prime so test next a
            while j < s and x != n − 1:
                x = x^2 mod n
                  # if x^2 = 1 mod n but x != +−1 mod n then n is not
prime
                if x == 1:
                    return False
                j = j+1
            # if a^r != 1 mod n and a^2jr != −1 mod n for every j in
(0,s−1) then n is not prime
            if x != n − 1:
                return False
    return True
```

Write a script performing the Miller-Rabin test. You can test it with the following numbers:

```
11072694125810661316518244826985010223678662935445814998022889497241657
5667359  (prime)


81116712346948063995695516399923130132965396377321265571871460960105500
481321   (composite)
```

If the script is correct, start the key generation. Choose a key length of 1024 bits. This is not very secure up to today's standards but will avoid having excessively long numbers. Then, you should try generating random prime candidates of 512 bits until you obtain two different prime numbers. Those are p and q. You have $N = p \cdot q$. For the public key, choose a random value $e \leq 2^{16} + 1$ and test that e is relatively prime with $phiN = (p-1)(q-1)$. The condition $e \leq 2^{16} + 1$ guarantees that the public key is a small number to reduce encryption complexity.

Testing this requires computing GCD(e, phiN), that can be done via the Euclidean algorithm. Since later we will also need the inverse of e modulo $phiN$, it is a good idea to directly implement the extended Euclidean algorithm. A convenient recursive version is shown here:

```
# returns (g,x,y) where g = GCD(a,b) and x,y verify x*a + y*b = g
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)
```

This exploits the fact that if x * (b % a) + y * a = g = gcd(b % a, a), then we have (y – (b // a) * x) * a + x * b = g = gcd(a, b). (Just observe that x * (b – (b//a)*a) = x * (b % a)).

Continue generating random e values until you find an e that is relatively prime with $phiN$. If you call (g, x, y) = egcd(e, phiN), when g == 1 you also have that x % phiN is the inverse of e modulo $phiN$. Call this value d.

The public key is the pair (N, e). The private key is the pair (N, d).

Keep the private key secret, publish the public key on the public chat together with your identifier, e.g., your group number. Now, everybody can send you encrypted messages that only you can decrypt. For decrypting a ciphertext c, just compute:

$$m = c^d \bmod N$$

And map the obtained m to a string using decodeText.

Diffie-Hellman key exchange

Encrypting long messages using RSA can be quite cumbersome. A more practical solution is to share a secret key using a key exchange protocol and then use the shared key with a conventional symmetric encryption scheme.

Generate a key pair for Diffie-Hellman. You already have public common parameters in AISC_03.py: p = pDH, g = gDH, q = qDH. For generating the private key, choose a random integer x between 1 and $q - 1$. This should be done securely, so use an instruction like:

```
x = int.from_bytes(os.urandom(256), byteorder='little')
```

This generates a random integer of 256*8 bits. Check that it is in the right range. Then, the public key will be:

$$pk = g^x \bmod p$$

You can publish your public key with your identifier. If you want to communicate with another group of students, get the group's public key pk_2, then compute the shared key as:

$$k = (pk_2)^x \bmod p$$

Extract from k the least significant 128 bits and convert them to a 16-byte bytearray:

```
key = (k & ((1 << 128) - 1)).to_bytes(16, byteorder='little')
```

This will be the shared key to be used with AES. The script AISC_03.py provides two functions to encrypt and decrypt messages using AES in CTR mode. To encrypt a string s convert it to a bytearray and pass it to the encryption function:

```
plaintext = s.encode('utf-8')
(iv, ciphertext) = encryptAESCTR(key, plaintext)
```

To decrypt a ciphertext and iv, call the decryption function and decode the bytearray:

```
plaintext = decryptAESCTR(key, iv, ciphertext)
print(plaintext.decode('utf-8'))
```

If you want to write iv and ciphertext on the public chat, it is better converting them in some readable characters. Use base64 encoding:

```
first_string = base64.b64encode(iv).decode("utf-8")
second_string = base64.b64encode(ciphertext).decode("utf-8")
```

For decoding a received ciphertext:

```
iv = base64.b64decode(first_string)
ciphertext = base64.b64decode(second_string)
```

Complexity of RSA and Diffie-Hellman

Encrypt and decrypt the file text.txt using RSA with a key of 1024 bits and measure the time required by the different steps: key generation, encryption, decryption. You can use time.time(), see the example on AISC_03.py.

Then, measure the time required by Diffie-Hellman key generation, shared key computation, AES encryption and AES decryption of the same text file.

Questions (Answer these in your report)

Report the RSA-encrypted message that you sent to the instructor and the answer that you received.

Report $N,\ p,\ q,\ e,\ d,$ values of your RSA key pair.

Report the average time that it takes your script to generate a valid RSA key pair and the time required to generate a valid Diffie-Hellman key pair and shared key.

Report the time required by RSA encryption and decryption and compare this with the time required by AES. Comment the results.

Assume that in an encrypted conversation one of the parties always responds with either "Yes" or "No". Do you see any difference between the case of RSA encryption and the case of AES encryption after key exchange? Motivate your answer.

Assume that you intercept a communication in which one of the parties always responds with either "Yes" or "No". Is this communication really confidential? Motivate your answer.

Bonus question: if you think that the above communication is not confidential, describe how you would modify encryption to achieve confidentiality.

Bonus question: in RSA, if you multiply a ciphertext c by a factor r modulo N you will obtain the encryption of the original message m multiplied by r modulo N. How could you exploit this to recover a message sent to the instructor by another group?