Computer Science Department
University of Crete

CS458: Introduction to Cryptography
Assignment 3 - Asymmetric Cryptography

Deadline: 7/1/2024

Introduction

The main purpose of this assignment is to offer you the opportunity to get familiar with asymmetric encryption and its various applications, as well as attack on them. Through these exercises you should be able to understand some of the difficulties of correctly using RSA.

Setup

When working on this assignment, remember to activate your Python virtual environment, that you were instructed to install in the first assignment, if you don't want Python modules to be installed system-wide:

```
cd hy458
source env/bin/activate
```

Make sure that the Python version you are using is at least 3.8 since the provided code uses type annotations.

For this assignment you will have to use the "cryptography" Python library:

```
pip install cryptography
```

Follow the libraries documentation for implementing the encryption and decryption processes: Asymmetric Algorithms-Cryptography

Also, for this assignment you will need the "gmpy2" Python library. This provides functions for operating on large numbers. To install it, run the command:

```
pip install gmpy2
```

If this fails, read the library documentation about the Installation.

Finally, you can use the "argparse" Python module to parse any necessary command line arguments.

Any data used inside the program should be represented as bytes. If you need to handle ciphertext or other input/output in the command line, you can convert it to the hexadecimal representation of the bytes. **You have to handle any conversions yourself!**

Also, your code should be able to be integrated into any system without changes. Use functions with appropriate parameters and minimize top-level code. For example, you can create a main function that is called as:

```
import sys

def f():
 print(‘‘Hello world’’)

def main(args):
 print(args)
 f()

if __name__ == ‘‘__main__’’:
 main(sys.argv[1:])
```

As you can see very little code runs on the top-level. Everything is wrapped in functions making it easy to integrate it in test files or larger systems.

Unless padding is explicitly ignored, use OAEP. For the label parameter of OAEP use None, for the hashing parameter of OAEP use SHA256.

Before starting, look into the provided code for hints. There are a few functions meant to help you with your implementation.

1. Digital Signatures: Basic Functionality [15%]

a. Create a Python function that, given a message and a *private* key, will calculate the digital signature of the message. The key generation should not be handled by this function, but the keys will be Cryptography's RSA key objects. This means that you can use functions and fields those key objects provide. For the generation of the digital signature you can use SHA256. For any necessary encryption you should implement RSA without padding. To implement RSA encryption without padding, do the following:

   (a) Convert the message from bytes to an integer
   (b) Raise the message (as an integer) to the private exponent modulo n
   (c) Convert the result back to bytes.

   Do not use any ready-made function for generating digital signatures. Notice that the message may or may not be encrypted. You don't have to handle the encryption of the message.

b. Create a Python function that, given a message, its signature and a *public* key, will verify the message matches the signature. The function should return a boolean, `True` if the message matches the signature, `False` otherwise. As before, the key generation should not be handled by this function, but the keys will be Cryptography's RSA Key objects. This means that you can use functions those key objects provide. For the verification of the digital signature you can use SHA256. For any necessary decryption you should use RSA. To implement RSA decryption without padding, do the following:

   (a) Convert the message from bytes to an integer
   (b) Raise the message (as an integer) to the public exponent modulo n
   (c) Convert the result back to bytes.

   Do not use any ready-made function for verifying digital signatures. Notice that the message may or may not be encrypted. You don't have to handle the decryption of the message.

c. Create a Python function to verify that both generation and verification work, for example by signing a message with the first function and then attempting to verify the signature. Also, test that a message not matching the one that generated the signature will not be accepted. Do this in two ways:

   i. Provide a message for signing and an entirely different message for verification.
   ii. Provide a message for signing and before verifying it change a single byte.

   This function should accept a private key (and from that extract the public key, see Cryptography's RSAPrivateKey documentation) and the message to be signed.

The script will accept the following command line arguments and pass them to the test function:

- -m: The message encoded in hexadecimal format
- -prv: The private key file path, in PEM format. You can use Cryptography's **load_pem_private_key** to help you. The key file is not encrypted.

2. RSA, Fermat Factorization: Breaking Confidentiality [15%]

For this exercise there is some code provided already. The file utils/key_generator.py implements the **fermat_vulnerable** key generation function. This creates and returns a public key with an n that is vulnerable to Fermat Factorization. This means that the factors are close to the square root of the modulus n. Your task is to factor n, recover the private key and use that to decrypt a message encrypted with the public key.

a. Create a function to implement the Fermat Factorization algorithm. You can consult the relevant lecture slides, Wikipedia article or other sources for the algorithm details. The function will accept the number that will be factored using the algorithm and it will return a pair of integers, the factors of n.

b. Create a Python function that, given a vulnerable public key, will recover the private key corresponding to the public key. To do this, you should follow the following steps:

   i. Factor the modulus n of the public key using the function in 2.a.
   ii. Using the factors of n calculate φ(n)
   iii. Use φ(n) and e (the public exponent) to calculate d (the private exponent).
   iv. Use Cryptography's partial key-handling methods to calculate the rest of the values necessary for creating a private key (hint: one of them is called iqmp). This step is to help you, so you don't have to implement RSA decryption yourself. If you want, you can do the calculations yourself, it is pretty straightforward.
   v. Combine the values calculated in the previous step and the public numbers to create an instance of Cryptography's RSAPrivateNumbers and use that to create a private key (RSAPrivateKey).
   vi. Return the private key.

c. Create a main function that will parse the command line arguments, call `fermat_vulnerable` to get a vulnerable public key and pass that key to the function in 2.b. Encrypt the message provided using the public key. Use the key returned by the function in 2.b to decrypt the encrypted message. Print the plaintext in hexadecimal format and save the private key to the correct path.

The script will accept the following command line arguments, parsing them in the main function:

- -m: The plaintext message encoded in hexadecimal format.
- -prv: The file path where the private key will be saved, in PEM format. Use Cryptography's **private_bytes()** to help you.

3. RSA Fermat Factorization: Forged Signatures [10%]

For this exercise there is some code provided already. The file utils/key_generator.py implements the **fermat_vulnerable** function. Your job is to factor n, use the gained knowledge to recover the private key and use that to forge the signature for a message.

a. Create a Python function that, given a message and a public key, will modify the message by changing the first byte to "G". Then, use the function from exercise 2.b to retrieve the private key from the public key. Use the functions from exercise 1. to create the signature for the modified message. Return the modified message and its signature.

b. Create a main function that will parse the command line arguments, call `fermat_vulnerable` to get a vulnerable key. Pass the message and the public key to the function in 3.a to get the modified message and its signature. This simulates a man-in-the-middle. Confirm that the signature of the modified message is still considered valid, that is it's verified by the public key. This means that the signature has been forged. **Which part of the CIA triad has been breached?**

The script will accept the following command line arguments, parsing them in the main function:

- -m: The original message, encoded in hexadecimal format.

4. RSA MITM: Whose key is that [10%]

In this exercise you will modify some given code in file **question4.py** to simulate a Man-In-The-Middle (MITM) attack. Add your code in the sections marked with **WRITE YOUR CODE HERE!!**. You shouldn't need to modify anything else. In this scenario, there is a client communicating with the server through a communication channel. You are the attacker, situated on the communication channel. Therefore, you can hear and alter the messages transmitted over the wire, after they are sent from the sender and before they arrive on the receiver. Your job is to eavesdrop on and modify the data sent by the client to the server. The data are encrypted using RSA. To mount the attack, you have to do the following steps:

a. When the client uses the communication channel to request the server's public key, the attacker would provide their own public key instead. This way, the attacker will be able to decrypt the data using their private key. Because the client cannot communicate with the server, they cannot realize that the key has been changed. Save the attacker's key in a variable to be able to access it later.

b. Store the public key published by the server in a variable. When the client tries to communicate with the server it will use the key it thinks is correct. However, this is not accurate because the key has been intercepted and swap for the attacker's. Use the attacker's private key to decrypt the message sent by the client (which has been encrypted using the attacker's key).

c. Demonstate that the attacker can access and modify the data. To do this print the decrypted message and add some bytes at the end.

d. Re-encrypt the data using the server's public key and send them to the server using its `recv` method. The server is none the wiser the data have been accessed and tampered with.

On this exercise **only**, you can use plaintext (instead of hex) when reading from the command line or when printing the data.
Is this attack possible on the Internet today? How do we defend against it?

5. RSA: Blind signatures [25%]

For this exercise you will implement a Blind Signature scheme based on RSA. This scheme involves two parties, the user that wants to obtain signatures for their message and the signer. There is also the recipient of the message, who will get the message and its signature and attempt to verify it. After the communication between the user and the signer is complete, the user has obtained a signature without the signer learning anything about the message. They can then submit the message and its signature to the recipient and the latter will use the signer's public key to verify the signature. Now, given that the signer has verified some property about the message and its sender, e.g. this sender is authenticated, has sent only one message, etc., the recipient can be sure that the property now holds.

- Create a class Signer that will take on the role of the signer. On initialization the class objects will create (and store) a private key with a 2048-bit modulus and 0x10001 as the public exponent. The class will provide the following instance methods:
  - `get_public_key(self)`: This will return the public key corresponding to the signer's (this object's) private key.
  - `sign(self, blinded_message: int)`: This will take the blinded message as a parameter and raise the message (as an integer) to the private exponent modulo n.
- Create a class User that will take on the role of the user who wants their message signed. On initialization the class objects will take a Signer object. The class will provide the following instance method:
  - `get_signed_message(self)`: This will create a random message and return that and its signature. Signing the message itself produces various problems. First, in addition to the unblinded message and signature, the blinded message and signature are also considered valid. Second, the message might be larger than the public modulus N. To solve these, a solution is to blind-sign a cryptographic hash of the message, not the message itself. Do the following steps:
    a. Hash the random message using SHA256.
    b. Convert the hash from bytes to an integer, let that be m.
    c. Select a random value r, less than N and relatively prime to N.
    d. Raise r to the public exponent (mod N) and multiply that with m (mod N). Let that be m'. This is the blinded message.
    e. Call the Signer object's `sign` method with m' to get the blinded signature, s'.
    f. Calculate the modular multiplicative inverse of r (mod N) and multiply it with s' (mod N) to get the unblinded signature as an integer.
    g. Convert the unblinded signature to bytes.
    h. Return the unblinded signature and the message, as bytes.

- Create a class Verifier that will take on the role of the recipient. On initialization the class objects will take a Signer object. The class will provide the following instance method:
  - `verify(self, message:bytes, signature:bytes)`: This will take the message and the signature and verify that they match. For the verification:
    a. Hash the message using SHA256.
    b. Convert the signature from bytes to an integer, let that be s'.
    c. Raise s' to the signer's public exponent (mod N). As shown in the lecture, this should be the message that was submitted to the signer (in this case it would be its hash). Let that be m.
    d. Convert the m from integer to bytes, and compare it with the hash.
    e. Return the result of this comparison. If True, it means that the message was successfully verified, proving the signature was valid.

- Create a main function that will test and showcase your implementation:
  a. Create a Signer object.
  b. Create a User and a Verifier object, passing them the Signer object from step a.
  c. Call User's `get_signed_message` method to get a message and its signature.
  d. Call Verifier's `verify` method:
     (a) With the message and signature as returned in step c. This should return True.
     (b) With a different or damaged message and the signature as returned in step c. This should return False.
     (c) With the message returned in step c. and a damaged signature. This should return False.
     (d) With a damaged message and a damaged signature. This should return False.

You don't have to handle user verification, assume that the Signer will sign anything send to them.

6. Side-channel attacks: Power analysis [25%]

For this exercise there is some code provided already. The file utils/power_analysis.py implements the **VictimComputer** class. This class simulates the computer receiving the encrypted message. It provides the following functions: get_public_key(), decrypt(ciphertext: bytes). get_public_key returns the public key (RSAPublicKey) that will be used for the encryption. decrypt takes the encrypted message, decrypts (but doesn't return) it and returns a list holding the samples of electrical power used by the CPU of the computer running the decryption (in Watts). As seen in the textbook, one might use this to derive information about the private key. Your job is to implement a function that will retrieve the private exponent from the power series. You can assume the following:

- While the power required may fluctuate, when the ALU is doing multiplication or squaring it uses at least 2W. While it is doing neither it uses less than 1W. If an exponent bit has the value 0, only a squaring is performed.
- If an exponent bit has the value 1, both a squaring and a multiplication is performed.
- Each single operation (either squaring or multiplication) takes at least 20 samples and at most 30.
- Squaring and multiplication takes longer than only squaring since it consists of two operations.
- In the case of squaring and multiplication there is at least 1 and no more than 5 samples of idle between the squaring and the multiplication.
- Between operations (square and multiply or square only) there is a small cooldown interval of at least 10 samples. Then the power will be less than 1W.
- The power trace will start with a few samples of idling.
- The power trace only includes samples from the decryption, i.e. the core is single-threaded.

To allow its own implementation, VictimComputer also holds the private key of the encryption internally. **You are not allowed to use that in your final implementation, only for testing before submission. Implementations using the private key from VictimComputer will be considered incorrect.**

a. Create a Python function that given a power trace will combine your knowledge of the square-multiply algorithm and the information given above to retrieve the private exponent of the communication. Be careful: According to the square-and-multiply algorithm the MS bit doesn't cause any operation so it doesn't appear in the power trace. Of course, the MS bit must always be 1. Remember to add it at the beginning of the private key.

b. Create a Python function that given the private exponent d and the public key will use Cryptography's rsa_recover_prime_factors to retrieve the prime factors p, q of n. Then, use these and Cryptography's partial key-handling methods to calculate the rest of the values necessary for creating a private key (hint: one of them is called iqmp). Again, this step is to help you, so you don't have to implement RSA decryption yourself. Return the private key.

c. Create a main function that will:

   i. Instantiate an VictimComputer object, let that be victim
   ii. Call victim.get_public_key to get the public key
   iii. Use the public key to encrypt a random message, composed of words in wordlist-1000.txt
   iv. Call victim.decrypt with the ciphertext to get the power trace
   v. Use the function in 6.a. to analyze the power series to retrieve the private exponent

vi. Use the function in 6.b. to get the private key from the public key and the private exponent

vii. Decrypt and print the message using the private key from step vi.

Submission

- Included with the code implementing the questions above, you should provide a README or PDF report file with your answers to the questions in bold above and explanations for your implemented or unimplemented parts.
- Submissions will be done through eLearn, you will be notified when they open.
- This assignment is an individual creative process and students must submit their own work. You are not allowed, under any circumstances, to copy another person's work. You must also ensure that your work won't be accessible to others.
- You are encouraged to post any questions you may have in the eLearn forum. If, however, you believe that your question contains part of the solution or spoilers for the other students you can communicate directly with the course instructors at hy458@csd.uoc.gr.
- Don't ask all your questions at the last moment, eLearn takes a while before sending them out to the course instructors.

References (accompanying files)

[1]. Square-multiply.pdf: Original text from the Understanding Cryptography textbook by Christof Paar and Jan Pelzl, Springer 2010.
[2]. Side-channel.pdf: Modified text, based on the Understanding Cryptography textbook by Christof Paar and Jan Pelzl, Springer 2010.