THE UNIVERSITY OF BRITISH COLUMBIA
Electrical and Computer Engineering Department

CPEN 442 **Introduction to Cybersecurity** **Winter Term 1, 2024**

Assignment 3: RSA and Hash Functions

Due date: **Friday, October 4, 2024, 11:59 pm**
Hard deadline: **Sunday, October 6, 2024, 11:59 pm**

Please use Piazza for questions and clarifications. Instructor and TA help will be provided only until 2pm on the same day as the *due date* above. This assignment has a 48-hour automatic deadline extension to account for unexpected issues when uploading the assignment. You are allowed to submit the assignment until the *hard deadline* date above, with no penalization. No exceptions after this date will be made!! If you have a long-term issue that will prevent you, or your group, from submitting the assignment, let the instructor know *well before the due date*.

Total marks: 100 pts
TA: Mohammed Elnawawy `[mnawawy@student.ubc.ca]`

Doing the assignment: This assignment can be done either **individually** or in **groups of two**. Regardless of what you do, you must join a group set in Canvas (check the `People` section, then `A3 group`). The group can be just yourself.

What to submit:

- A PDF file called `a3.pdf`, with the answer to Question 3.3 (everything else is coding).
- A Python file `a3.py`; please see the provided template and do not change the function headers.

1 Basic modular arithmetic in Python

In the first part of this assignment, you will work with modular arithmetic. To compute the modulo operation in Python, you can simply use `%`. For example:

```
# Example from slide 79
n = 16
print(12 * 3 % n)
print(11 * 3 % n)
print(12 * 11 * 3 % n)
```

Before you implement Textbook RSA, you need two basic functions: one that computes modular exponentiations efficiently, and one that computes the modular multiplicative inverse.

1.1 Efficient modular exponentiation

For small numbers, modular exponentiation in Python is very simple:

```
result = x**y % n
```

However, when the numbers get big, this becomes very inefficient.

There is a popular algorithm to compute the modular exponentiation where the computational cost is $O(\log b)$, where b is the number of bits of the exponent.

The idea is the following: assume we want to compute x^y and $y = 11$. If we write the exponent in binary (11 in decimal is 1011_2 in binary), we can write the exponentiation as $x^y = x^8 \cdot x^2 \cdot x^1$. In other words, we can compute the exponentiation by representing the exponent in binary, iterating through the bits, and multiplying by the relevant power of x when the bit is 1. See the pseudocode in Algorithm 1. Note that we can apply the modulo operation at every step, to make sure numbers do not get unreasonably big. This does not affect the outcome.

[5 pts] Question 1.1: Program the function `mod_pow` in `a3.py` following Algorithm 1. It is up to you to verify the implementation is correct. We will test this with large numbers. In particular, we will generate p and q as random 1024-bit primes, multiply them to get n, and raise random numbers to random exponents (all modulo n). To generate random primes, check Pycryptodome's documentation (look for the function `getPrime`). Your function should output results pretty fast (e.g., usually around 1-2 seconds). If you try to do it with "`x**e % n`", you will be less lucky.

Algorithm 1 Binary algorithm for modular exponentiation ($x^y \pmod n$)

```
procedure MOD_POW(x, y, n)
    Initialize result: r = 1
    while y > 0 do
        if y is odd then
            r = r · x (mod n)
        end if
        Shift y to the right one bit.
        x = x · x (mod n)
    end while
end procedure
```

After you finished programming `mod_pow`, you should also copy it inside `utils.py`. This script imports the function from a different file `a3secret.py`, which we cannot reveal to not spoil the solution of the previous question.

1.2 Computing modular multiplicative inverses

Next, you will implement the modular multiplicative inverse, as seen in the classroom. Inside `utils.py`, you already have an implementation of the Extended Euclidean algorithm. This algorithm receives two integers (a and b) and returns three integers: the greatest common divisor of a and b, an integer x, and another integer y, such that:

$$a \cdot x + b \cdot y = \gcd(a, b)\,.$$

Using this function, implement the function `mod_inv(x, n)`, which computes the modular multiplicative inverse of x modulo n. Check slide 72 in module 2 for a reminder of how to use the Extended Euclidean algorithm to compute the modular inverse. A few requirements:

1. The modular inverse must be a value in the interval $[0, n-1]$.
2. If the modular inverse does not exist, your function must raise an error.

[5 pts] Question 1.2: Implement the function `mod_inv` inside `a3.py` following the specifications above.

Verify that your function works with large numbers (e.g., n is around 2048 bits long, and x is a random number modulo n).

2 The RSA cryptosystem

Recall the process for encrypting and decrypting in RSA. To encrypt, we have a public key (e, n) and a message M (which is a positive integer smaller than n). The encryption of M is simply $C \equiv M^e \pmod{n}$. To decrypt a ciphertext C (also a positive integer smaller than n), we have a private key d, and we compute $D \equiv C^d \pmod{n}$.

Algorithm 2 Textbook RSA encryption and decryption

procedure ENCRYPT(M, e, n)
  Return $C \equiv M^e \pmod{n}$
end procedure
procedure DECRYPT(C, d, n)
  Return $D \equiv C^d \pmod{n}$
end procedure

2.1 Trivial attack when one of the prime numbers leaks

Instead of implementing RSA directly, you'll do something indirect: let's consider a trivial attack that happens when Eve learns p.

Assume Alice has published her public key (e, n), where $n = p \cdot q$ for two large primes p and q. To encrypt a message for Alice, Bob uses the following *hybrid approach*:

```
# Bob creates message for Alice
M_str = "Hello, world!"
# Encrypt message with random AES key (IV is all-zero bytes)
aes_key = randbytes(AES.block_size)
cipher = AES.new(aes_key, AES.MODE_CBC, iv=bytes(AES.block_size))
C_aes = cipher.encrypt(pad(M_str.encode('ascii'), AES.block_size))
# Encrypt AES key with Textbook RSA (params: M, e, n)
aes_key_as_int = int.from_bytes(aes_key, byteorder='big')
C_rsa = textbook_rsa_encrypt(aes_key_as_int, e, n)
# Here, Bob would send Alice (C_aes || C_rsa)

# Your attack: Eve intercepts the ciphertext and knows public parameters (e,
    n), plus p:
recovered_text = a3.attack_q21(C_aes, C_rsa, e, n, p)
# Checking the attack correctly recovered the plaintext:
assert recovered_text == M_str
```

[10 pts] Question 2.1: Implement the function `attack_q21(C_aes, C_rsa, e, n, p)`

which takes as inputs the messages Bob sent to Alice (`C_aes` and `C_rsa`), Alice's public key (`e` and `n`), and one of the primes `p`, and returns the recovered text as a string.

To test it, use the parameters in `q21_parameters.txt` (you can just copy-paste them in a Python script to do a test). Note that the AES ciphertext is stored in Base64 encoding in the text file; you must deocode it before passing it to the function. Your function `attack_q21` should output a readable string.

Fun fact: in practice, while p and q are large primes chosen at random, many implementations of RSA use $e = 65\,537$. This is because this number is prime (so $\gcd(e, \varphi) = 1$), and its binary representation only has two 1 bits: $e = 2^{16} + 1$. This means that the encryption using this e is extremely fast (remember the `mod_pow` function you wrote before). The RSA cryptosystem is still secure even if e is not chosen as random, since n is different every time, and the adversary cannot guess d unless they are able to factor n.

2.2 Naive RSA signatures

Next, we will see a naive approach to compute signatures using Textbook RSA. We saw that using Textbook RSA for encryption/decryption is insecure; the same is true for using it for signatures, so you should never use this in practice.

In RSA, the key d is private, while e and n are public. To sign a message M, we can "decrypt" the message first, i.e., compute the tag $T \equiv M^d \pmod{n}$. Note that we are performing a decryption operation on a plaintext M, so it will lead to some random-looking tag T. However, we are the only party that knows d, so we are the only ones that can perform this operation. This tag T is the signature. If we share (M, T) with someone, they can verify the signature by checking that $T^e \equiv M \pmod{n}$ (using e and n as the public verification key). If this check is successful, then it means that T is a valid signature of M.

Algorithm 3 Textbook RSA signatures and verification

procedure SIGN(M, d, n)
    Compute tag $T \equiv M^d \pmod{n}$
    Return (M, T)
end procedure

procedure VERIFY$((M, T), e, n)$
    if $M \equiv T^e \pmod{n}$ **then**
        Return 1
    else
        Return 0
    end if
end procedure

Let's see some of the vulnerabilities of these signatures. For these exercises, assume you only know the public verification key, i.e., (e, n). We will see that it is possible to *forge* signatures, i.e., to create valid (M, T) pairs, even when we do not have the secret key d.

[10 pts] Question 2.2: Program the function `forge_signature1(e, n)`, which returns a message and a tag (M, T) such that T is a valid tag (signature) for M. Every call to this function should provide a fresh random pair (M, T), i.e., do not return always the same (M, T). There are no requirements about the format of M and T, other than they are numbers modulo n and that $M \neq 1$. Note that you do not have d.

You can use `evaluate_forge_signature1` in `utils.py` to verify your implementation.

[10 pts] Question 2.3: Program the function `forge_signature2(M1, T1, M2, T2, e, n)`, which receives the valid message-signature pairs (`M1`, `T1`) and (`M2`, `T2`) and returns a valid signature T_3 for the message $M_3 = M_1 \cdot M_2$.

You can use `evaluate_forge_signature2` in `utils.py` to verify your implementation.

[18 pts] Question 2.4: You want Alice to sign message M, which is a numerical encoding of "Transfer all of Alice's money to Eve", but Alice obviously refuses to do so. However, Alice is willing to sign *any other* message $M' \neq M$. Program the function `forge_signature3(e, n, sign)`, which receives Alice's public key (`e` and `n`), as well as access to Alice's `sign(M)` function, and returns a tuple (`M`, `T`). The message `M` must be the encoding of "Transfer all of Alice's money to Eve", and `T` must be a valid signature for this. You can use the functions `str_to_int` and `int_to_str`, inside `utils.py` to convert strings to integers (up to 256-byte long). If your integer does not fit into a 256-byte string, then you are probably doing something wrong.

We will evaluate this function by running the following script:

```
def evaluate_forge_signature3():
    alice = utils.Alice() # instantiate Alice
    e, n = alice.get_public_key() # get her public parameters
    M, T = a3.forge_signature3(e, n, alice.sign) # forge signature
    alice.check_q24_correctness(M, T) # verify you forged the right signature
```

The last function call must print "`You've successfully forged the signature!`". Check the code of the `Alice` class under `utils.py` for more information. We will change Alice's key-pair for grading.

3 Hashing and Authentication

In the classroom, we saw that a strong cryptographic hash function must meet three properties: pre-image resistance, second pre-image resistance, and collision resistance.

In `utils.py`, you have the function `my_hash`. This function computes the SHA-256 hash of a `bytes` input and keeps the first 3 bytes only as a "new hash".

[10 pts] Question 3.1: Program the function `find_collision(hash)`, which receives a hash function `hash` (h) and outputs two inputs (x and x') such that $h(x) = h(x')$.

[10 pts] Question 3.2: Program the function `find_preimage(hash, digest)`, which receives a hash function `hash` (h) and a `digest` (y), and finds the input (x) such that $h(x) = y$.

[7 pts] Question 3.3: This one is just for `a3.pdf`: run `find_collision(my_hash)` and `find_preimage(my_hash, random.randbytes(3))`, computing their run times. In `a3.pdf`, report the runtimes that you got for each function, and a *brief* explanation of why you see this runtime difference.

You can try and modify `my_hash` so that it outputs 4 bytes, and then find both a collision and a pre-image. You will see that the time it takes to solve the functions increase significantly (the increase is exponential). For reference, SHA-256 returns a 256-bit digest (32 bytes). Finding a pre-image, even if we controlled all the computers in the world, would take more than the age of the universe.[1]

3.1 Encrypting a hash to provide authentication?

Warning: this question takes significantly more time than the others. Do at your own risk. It is only 15% of this assignment (roughly, around 1% of the total course grade).

This year, someone asked if the message format $E_K(M \| h(M))$ provides integrity and authentication. Intuitively, it seems that using such a format is OK, since Eve does not know K, and it is not possible for her to modify the ciphertext in a way that the format of the resulting plaintext $(M \| h(M))$ is still valid.

However, it is not a simple as that. Consider a case where Alice and Bob communicate using the message format above:

1. Alice and Bob use AES-CBC as their encryption algorithm.

[1] Unless someone finds a way of cracking SHA-256 faster than brute force, of course.

2. They use a strong hash function: SHA-256.

3. When Alice wants to send a message M to Bob, she pads the message to a multiple of the block size, concatenates it with its hash, and encrypts it using AES-CBC with a randomly generated IV.

4. Then Alice sends Bob the ciphertext and IV: $E_{K,IV}(M \| h(M)) \| IV$.

5. To check for integrity and authenticity, Bob decrypts the first part of the message with the secret key K and using the given IV, and checks that the last 32 bytes of the plaintext are indeed the hash digest of the message. Then, he unpads the message.

You can find the implementation of Alice and Bob in the `AliceAndBob` class in `utils.py`. When an object of this class is instantiated, a random AES key is generated. Then, this class defines three methods:

1. The method `send_message(M: bytes)` receives a message to be sent as a `bytes` object, and returns the ciphertext and IV (following the step outlined above).

2. The method `receive_message(C: bytes, IV: bytes)` receives a ciphertext and an IV and does the integrity/authentication check explained above. If the message passes the test, then the function prints "`Message accepted: `", followed by the message.

3. There is an additional method, called `send_message_from_eve(M: bytes)`. This function creates a new message to warn the other party that Eve said something:

```
def send_message_from_eve(self, M_eve: bytes):
    M = b"Eve said this, do not trust her: '" + M_eve + b"'"
    return self.send_message(M)
```

This seems harmless: if integrity and authentication work as they should, it should be safe to craft a message like this. Turns out this is not true when we implement integrity with a regular hash function (instead of using a MAC).

[15 pts] Question 3.4: Program the class `Eve` inside `a3.py`, with methods `craft_message` and `modify_ciphertext_and_iv`. The first method, `craft_message`, receives a string `M_evil` and returns a bytes object. Eve sends this bytes object to Alice, who warns Bob that Eve is crafting such message by calling `send_message_from_eve`. However, Eve captures the ciphertext and IV, and modifies them before sending them to Bob using `modify_ciphertext_and_iv`. Show that Eve can make Bob believe that Alice said `M_evil`.

The grading of this question will follow this process:

```
def testing_alice_and_bob():
    aliceandbob = utils.AliceAndBob() # initialize Alice and Bob
```

```
eve = a3.Eve() # initialize Eve
M_eve = eve.craft_message("I hate you, Bob!!") # Eve crafts a message
C, IV = aliceandbob.send_message_from_eve(M_eve) # Alice warns Bob about the
    message
C, IV = eve.modify_ciphertext_and_iv(C, IV) # Eve modifies ciphertext and IV
aliceandbob.receive_message(C, IV) # Bob receives the modified message
```

If the attack is correctly implemented, this should print:

```
Message accepted: 'I hate you, Bob!!'
```

This should work for other ascii strings that we pass to `eve.craft_message` as well.

Good luck!

Fun fact: you might think that this "`send_message_from_eve`" is unrealistic. Why would someone encrypt some bytes that someone else generated? However, 1) the security of a cryptographic algorithm should not rely on what the content of the message is and 2) there are many real cases where an adversary has this potential "chosen plaintext attack". For example, someone (Alice) might choose to encrypt their email at rest, so that the email provider cannot read every single word in there. In that case, Eve can send an email to Alice, and Alice's program would automatically encrypt the email and store it in the database. The email provider could very well be Eve.