CO 487 - Assignment 3

Bilal Khan (b54khan)
bilal.khan@student.uwaterloo.ca

November 4, 2024

Contents

1 1 **1**
1.1 a . . . 2
1.2 b . . . 3
1.3 c . . . 3
1.4 d . . . 3
1.5 e . . . 3

2 2 **4**
2.1 a . . . 4
2.2 b . . . 4
2.2.1 a . . . 5
2.2.2 b . . . 5
2.2.3 c . . . 5
2.2.4 d . . . 5

3 3 **6**
3.1 a . . . 6
3.2 b . . . 6

4 4 **7**
4.1 a . . . 7
4.2 b . . . 8

5 5 **9**
5.1 a . . . 9
5.2 b . . . 9
5.3 c . . . 10

1 1

In this problem, you will be asked to create a small Python 3 script for encrypting and decrypting snippets of text. We will be using the cryptography library for Python 3, specifically the primitives in the "hazardous materials" layer of the cryptography library.1 You can read the documentation here: https://cryptography.io/en/latest/.

The starting code for this question is distributed as a Jupyter notebook called a3q1.ipynb which you can download from LEARN. You can then upload this notebook to the University of Waterloo Jupyter server at https://jupyter.math.uwaterloo.ca/. You can do all the programming for this question in the web browser; you do not have to install anything on your computer.

You should implement two functions: one for encryption and one for decryption. Each function will take as input a string (the message or ciphertext, respectively), then prompt the user for a password, perform any relevant cryptographic operations, and then return the result.

You should use the following cryptographic primitives to build an IND-CCA-secure authenticated encryption scheme using password-derived keys:

- AES-256 for encryption in cipher feedback mode,
- HMAC to provide integrity,
- Scrypt as a key derivation function,
- SHA3-512 as a hash function.

You can use any of the primitives in the Hazardous Materials Layer of the Cryptography package (no recipes).

1.1 a

Write your encryption and decryption procedures in pseudocode.

```
def encrypt(message, password):
    salt = random(16)
    IV = random(16)

    aes_key = scrypt.generate_key(password, salt)
    hmac_key = scrypt.generate_key(password, salt)

    message = message.utf8()

    aes = AES(aes_key, IV, CFB)
    hmac = HMAC(hmac_key)

    ciphertext = aes.encrypt(message)
    tag = hmac.tag(iv || ciphertext)

    return salt || iv || ciphertext || tag


def decrypt(ciphertext, password):
    salt || iv || ciphertext || tag = ciphertext

    aes_key = scrypt.generate_key(password, salt)
    hmac_key = scrypt.generate_key(password, salt)

    aes = AES(aes_key, IV, CFB)
```

```
    hmac = HMAC(hmac_key)

    if not hmac.verify(iv || ciphertext, tag):
        throw Error

    return aes.decrypt(ciphertext)
```

1.2 b

Implement your encryption and decryption procedures in the provided Jupyter notebook. If you need to include multiple values in an output, you might find it helpful to use tuples or JSON data structures. Submit your code through Crowdmark, as you would a normal assignment - as a PDF or screenshot of your Jupyter notebook. Make sure your code is readable and has helpful comments, as we will be grading it by hand, not executing it.

1.3 c

How many bytes of overhead does your encryption function add on top of the minimum number of bytes required to represent the message itself?

```
msg = "Hello, world!"

msg_len = len(msg.encode('utf-8'))
encrypted_len = len(encrypt(msg).encode('utf-8'))
print(f'{msg_len=}, {encrypted_len=}, {encrypted_len-msg_len=}')
# msg_len=13, encrypted_len=207, encrypted_len-msg_len=194
```

This code prints 194 bytes of overhead for the encryption function on top of the 13 bytes needed to represent the message itself.

1.4 d

For the Scrypt parameters you chose, what is the estimated runtime for a single execution? How much memory will be required for Scrypt to operate?

The estimated runtime is 0.1s from timing the encryption function. the memory cost is defined by scrypt params $n = 2^14$ and $r = 8$ and according to the scrypt docs, is $128 * n * r = 16MB$.

1.5 e

What is one mistake that someone could make when solving this question that could undermine security?

Using the same key ofr aes/hmac for example

2 2

In this problem, you will be cracking password hashes. You will have to write your own code to do this. We recommend using Python and its hashlib library, which includes functions for computing a SHA256 hash. For example, the following line of Python code will compute the SHA256 hash of a string s:

```
hashlib.sha256(s.encode()).hexdigest()
```

Please include all code used in your submission.

Suppose you are a hacker that is interested in gaining access to Alice's online accounts. You've managed to obtain the user databases from several websites, giving you knowledge of the hashes of several of Alice's passwords. All hashes in this question are SHA256.

2.1 a

The hash for Alice's password on example.com is

```
303b8f67ffbe7f7d5e74e9c2177cd1cabbc9dc53154199f540e1901591c7d5fa
```

example.com prepends salts to their passwords before hashing, as well as requiring that passwords be exactly 6 digits (0-9). The salt that was included with Alice's password is 19147384.

What is Alice's password? Submit any code you write along with your answer.

The password is 429933.

```python
import hashlib

def decrypt():
    hash = "303b8f67ffbe7f7d5e74e9c2177cd1cabbc9dc53154199f540e1901591c7d5fa"
    salt = "19147384"

    for i in range(1000000):
        password = f"{i:06d}"
        hashed = hashlib.sha256((salt + password).encode()).hexdigest()
        if hashed == hash:
            print(password)
            return
    return False

decrypt()
# 429933
```

2.2 b

On example.org, they have stricter security requirements. First, they prepend a 64-bit salt k1 to each password. The same salt k1 is prepended to each password, and is kept secret (in particular, it is not stored alongside the password in the database, so compromising the user database does not reveal the fixed secret salt). The concatenated salt and password is then hashed with a cryptographic hash function. Then, the result is encrypted with a modified version of AES that

uses a 64-bit key k2. Again, the same key k2 (which is not part of the database compromise) is used to encrypt every password. Finally, the result is stored in the password database alongside its corresponding username. More concisely:

```
c = AES.Enc(k2, H(k1 ||pw))
```

Suppose that, before you compromised the user database, you registered a few fake accounts on the server for which you know the plaintext password.

2.2.1 a

Is it feasible to carry out a exhaustive search to recover the salt and encryption key? Why or why not?

No. The salt is unknown and 64 bits and the key is unknown and also 64 bits, so a brute-force attack would need to check every possible salt for every possible key.

Registering a few ($<< 2^{64}$) fake accounts doesn't help because we would still need to check every possible salt for every possible key. The only case it might help is if it can help us check for any false positives from hash collisions.

2.2.2 b

Describe an attack which recovers both the key and the secret salt that takes only about $\sqrt{X}$ hash/encryption operations (or some small constant multiple thereof), where X is the number of operations required for a exhaustive search.

Given a known plaintext/ciphertext pair that we get from one of our fake registered accounts, compute and cache the hash of every possible 2^{64} salt with the known plaintext. Then, for every possible 2^{64} key, decrypt our known ciphertext with it and check if it exists in our cache. If it does, we've found the correct salt and key. Since the same salt and key are used for every passord, we have broken the password in $O(\sqrt{2^{128}}) = O(2^{64})$ time.

2.2.3 c

Would the same attack work if example.org first encrypted each password and then hashed it with their secret salt? Why or why not?

No, the original attack was possible because we could independenly undo/invert/decrypt the ciphertexts to get our intermediate value that we checked agaisnt the hashes. This is imossible here since we can't invert a secure hash function to get an intermediate value that we try decrypting.

2.2.4 d

Would you recommend using this method of password hashing? Why or why not?

No, it depends on the salt and key both never being leaked. If one is leaked then the attack becomes much easier (2^{64} ops instead of 2^{128}.) and requires all passwords to be rehashed (impossibel to be done automatically since we never store the passowrds and need to wait for users to login again to update the stored hashes.)

3 3

Variable-length input MAC schemes

In this question, we will explore attempts to build a MAC that handles variable-length inputs from MAC schemes that have fixed-length inputs.

Let $\mathcal{M} : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0,1\}^{256}$ be a secure MAC scheme that takes as input a 256-bit key and a fixed-length 256-bit message, and produces as output a 256-bit tag. While useful for small messages of fixed length, this MAC scheme cannot be used for longer messages without modification. Here are two attempts at such a modification. For each proposal, your task is to decide if the scheme is secure or not. If it is, prove it using the assumption that $\mathcal{M}$ is secure. Otherwise, propose an attack.

For the rest of this question, for the variable-input-length MAC scheme we are trying to construct, we will simplify to the setting where the input messages have a length that is a multiple of the block size, and ignore the issue of padding.

3.1 a

For a message m whose length is a multiple of 256, split it into blocks $(m_1, \ldots, m_n)$ each of length 256, and apply the following algorithm:

Algorithm 1: BIGMAC(k, m)

for $i = 1, \ldots, n$ **do**
  $t_i \leftarrow \mathcal{M}(k, m_i)$
end
return $t_1 \| \cdots \| t_n$

Insecure. You can reorder or duplicate blocks of the original message and the tag together. This will result in a valid tag for the new message. The fact that we apply the MAC to each block independently and concatenate the tag of each blobk together makes ti vulnerabel to reorderign or duplicating attacks.

3.2 b

For a message m whose length is a multiple of 256, split it into blocks $(m_1, \ldots, m_n)$ each of length 256, and apply the following algorithm:

Algorithm 2: MACNCHEEZ(k, m)

$t_0 \leftarrow 0^{256}$ **for** $i = 1, \ldots, n$ **do**
  $t_i \leftarrow \mathcal{M}(k, m_i \oplus t_{i-1})$
end
return t_n

Insecure, the attack is the same as for CBC-MAC. Choose a fixed length message of one block m_1 and t_1 be its oracle-queried tag. now request the tag t_2 for the message $m_2 = t_1$. Then t_2 is also the tag for the two-block message $m_1 \oplus 0$

4 4

We wanted to see if ChatCPT was a good cryptographer.2 We asked ChatGPT to construct an authenticated encryption scheme by combining an encryption scheme with a MAC. The following are some of the bad constructions it proposed, ignoring those that do not compute at all. Your task is to explain why each construction is not secure.

Let E_{k_E} be an IND-CPA secure symmetric key encryption scheme with secret key k_E, and let M_{k_M} be a secure (existentially unforgeable under chosen message attack, a.k.a. EUF-CMA) message authentication code with secret key k_M.

For each of the following authenticated encryption schemes:

(i) Write out the corresponding decryption procedure. If you want to indicate that the ciphertext is not valid, you can use the pseudocode "return error".

(ii) Demonstrate that the scheme is insecure by doing the following:

- Choose a secure symmetric key encryption scheme for E_{k_E} and explain why it is secure.
- Choose a secure MAC for M_{k_M} and explain why it is secure.
- Prove that the proposed authenticated encryption scheme is not secure (either it lacks IND-CPA security, or it lacks EUF-CMA security) when using your encryption scheme and MAC by explaining how to attack it.

Note that when we say "Choose a secure symmetric key encryption scheme" (or "Choose a secure MAC"), this means that you can assume there exists a generic secure symmetric key encryption scheme (or secure MAC), and you can either use it directly, or you can use the "degenerate counterexample proof technique" (Topic 2.1, slides 23-26) to build a second symmetric key encryption scheme (or MAC) from the generic one that remains secure (show this!), but conveniently has some strange property that helps you demonstrate the insecurity of the proposed authenticated encryption scheme.

4.1 a

"Cascading Encryption and MAC Structure":

Algorithm 3: $AEON(k = (k_E, k_M), m)$

$C \leftarrow E_{k_E}(m)$
$T_1 \leftarrow M_{k_M}(m)$
$T_2 \leftarrow M_{k_M}(T_1 \parallel C)$
return (C, T_1, T_2)

Assume that we are using a SKES that is IND-CPA secure and a MAC that is EUF-CMA secure.

This is insecure because macs are not required to ensure confidentiality. Computing the MAC of the plaintext and sending that as part of the ciphertext can leak information about the plaintext, so the overall authenticated encryption scheme fails IND-CPA.

Algorithm 4: $AEON_{dec}((k_E, k_m), (C, T_1, T_2))$

$m' \leftarrow D_{k_E}(C)$
$T_1' \leftarrow M_{k_M}(m')$
$T_2' \leftarrow M_{k_M}(T_1'||C)$
if $T_1 \neq T_1'$ ***or*** $T_2 \neq T_2'$ **then**
  return error
end
else
  return m'
end

4.2 b

"XOR-Dependent Combined Encryption and MAC":

Algorithm 5: $AETHER(k = (k_E, k_M), m)$

// select a uniformly random nonce of length equal to the plaintext
$N \leftarrow \{0,1\}^{|m|}$
$C \leftarrow E_{k_E}(m \oplus N)$
$T \leftarrow M_{k_M}(C \oplus N)$
return (C, T, N)

Algorithm 6: $AETHER_{dec}((k_E, k_M), C, T, N)$

$m' = D_{k_E}(m) \oplus N$
$T' = M_{k_M}(C \oplus N)$
if $T \neq T'$ **then**
  return error
end
else
  return m'
end

5 5

The rivalry between math students and engineers has advanced to the next stage. After CO 487 students successfully broke the Awesome Engineering Squad's Not Very Permute-y cipher a month ago, a splinter group called the Really Super Awesome Association of Engineering Students (RSA-AES) have taken over with their revolutionary use of public key cryptography. However, as everyone knows, engineers are very odd. So odd, in fact, that the Really Super Awesome Association of Engineering Students will only consider RSA-encrypted messages sent to them if those decrypted messages are odd (recall that messages for RSA are integers). To avoid angering people who do not get a response because their decrypted message is not odd, when the RSA-AES receives a ciphertext, they immediately decrypt it, then send a plaintext response back to the sender indicating whether they will be considering their message or not. (We call this interaction the "RSA plaintext parity checking oracle".)

Let (N, e) be the RSA public key of the Really Super Awesome Association of Engineering Students. Suppose you intercept a ciphertext, c^*, of a message $m^* \in [0, N-1]$ (with m^* odd), encrypted under (N, e).

5.1 a

Show that, if $m_1, m_2 \in [0, N-1]$ with RSA encryptions c_1 and c_2, respectively, then $RSA.Enc((N,e), m_1 \cdot m_2) = c_1 \cdot c_2 \mod N$.

$$\begin{aligned} RSA.Enc((N,e), m) &= m^e \mod N \\ RSA.Enc((N,e), m_1 \cdot m_2) &= (m_1 \cdot m_2)^e \mod N \\ &= m_1^e \cdot m_2^e \mod N \\ c_1 &= m_1^e \mod N \\ c_2 &= m_2^e \mod N \\ c_1 \cdot c_2 &= m_1^e \cdot m_2^e \mod N \\ &= RSA.Enc((N,e), m_1 \cdot m_2) \mod N \end{aligned}$$

5.2 b

Show how you can determine whether $m^* > N/2$ or $m^* < N/2$, by interacting with the RSA plaintext parity checking oracle. Justify mathematically why your approach works.

Use the property from a that $RSA.Enc((N,e), m_1 \cdot m_2) = (c_1 \cdot c_2) \mod N$. We can multiply m^* by 2 and then query the oracle to get c'. If c' is even, then $m^* < N/2$ and if c' is odd, then $m^* > N/2$.

$$\begin{aligned} RSA.Enc((N,e), 2 \cdot m^*) &= (2^e \cdot c^*) \mod N \\ (2 \cdot m^*)^e \mod N &= c' \end{aligned}$$

The product of two with any number will always be even, so $2 \cdot m^*$ will always be even. If $2 \cdot m^*$ is greater than N (a product of two primes, that is itself odd), then $2 \cdot m^* \mod N = 2 \cdot m^* - N$, which is an even number $(2 \cdot m^*)$ minus an odd number (N) and is therefore always odd.

Therefore, if c' will be even then $2 \cdot m^* < N$ or $m^* < N/2$. Oterwise, c' is odd, $2 \cdot m^* > N$, and $m^* > N/2$.

5.3 c

Show how you can efficiently recover all of m^*. Justify mathematically why your approach works. How many interactions with the RSA plaintext parity checking oracle do you need for your attack?

Hint: Binary search

Follow the same approach as in b, but instead of multiplying by 2, we can perform binary search in the range $[0, N)$ to find $m*$. Calculate the midpoint of the current range, query the oracle with it and use the result being odd or even to decide whetehr to search in the upper or lower half of the remaining range respectively. This terminates once we narrow it down to a single value. This will take $O(\log N)$ queries as is normal in binary serch.