C&O 487/687: Assignment #3 Solutions

Due date: November 4, 2024 (Monday)

If you are requesting an extension or accommodation because of a verification of illness form, self-declared student absence, or AccessAbility Services accommodation, see the "Missed assignment / extension policy and forms" instructions on the CO 487/687 LEARN site.

1. [10 marks] **Symmetric encryption in Python**

   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).

   (a) [5 marks] Write your encryption and decryption procedures in pseudocode.

   Solution. Encryption:

   i. Pick a random 16-byte salt for Scrypt
   ii. Run Scrypt with the given password, salt, and parameters $n = 2^{14}$, $r = 8$, $p = 1$, and produce a 64-byte output that's split into two 32-byte keys k_1 and k_2
   iii. Pick a random 16-byte initialization vector for AES-256 CFB mode
   iv. Encrypt the plaintext using AES-256 in CFB mode with k_1 and the selected IV
   v. Apply HMAC-SHA3-512 using key k_2 to the ciphertext and the iv (it's a good idea to authenticate the IV as well)
   vi. Output the salt, IV, ciphertext, and tag

   Decryption:

   i. Run Scrypt with the given password, salt, and same n, r, and p parameters, and produce a 64-byte output that's split into two 32-byte keys k_1 and k_2

[1] This portion of the library is labelled "hazardous materials" because it is easy for inexperienced users – and people who didn't do well in CO 487 – to use these building blocks incorrectly. Good cryptographic libraries often include all-in-one functions that are harder to use incorrectly. The Python 3 `cryptography` library has a package called "Fernet" recipes which are meant to be used by non-experts and are harder to misuse. If you are writing programs that cryptography in your career after CO 487, I recommend trying to use good libraries with good APIs for all-in-one recipes whenever possible to reduce the risk of human error.

ii. Apply HMAC-SHA3-256 using key k_2 to the ciphertext and the IV; reject if the computed tag doesn't match the given tag
iii. Decrypt the ciphertext using AES-256 in CFB mode with k_1 and the given nonce
iv. Output the plaintext

(b) [2 marks] 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.

Solution.

```
def encrypt(message):

    # encode the string as a byte string, since cryptographic functions usually work on bytes
    plaintext = message.encode('utf-8')

    # Use getpass to prompt the user for a password
    password = getpass.getpass("Enter␣password:")
    password2 = getpass.getpass("Enter␣password␣again:")

    # Do a quick check to make sure that the password is the same!
    if password != password2:
        sys.stderr.write("Passwords␣did␣not␣match")
        sys.exit()

    ### START: This is what you have to change

    # derive 2 keys from the password using Scrypt
    salt = os.urandom(16)
    kdf = Scrypt(
        salt=salt,
        length=64,
        n=2**14,
        r=8,
        p=1
    )
    key = kdf.derive(password.encode('utf-8'))
    key1 = key[0:32]
    key2 = key[32:64]

    # encrypt the plaintext
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key1), modes.CFB(iv))
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(plaintext) + encryptor.finalize()

    # authenticate the ciphertext (including iv)
    hm = hmac.HMAC(key2, hashes.SHA3_512())
    hm.update(iv)
    hm.update(ciphertext)
    tag = hm.finalize()

    # put it all into a JSON dictionary for easy processing
    output = {}
    output["salt"] = bytes2string(salt)
    output["iv"] = bytes2string(iv)
    output["ciphertext"] = bytes2string(ciphertext)
    output["tag"] = bytes2string(tag)

    return json.dumps(output)

    ### END: This is what you have to change

def decrypt(ciphertext):

    # prompt the user for the password
    password = getpass.getpass("Enter␣the␣password:")

    ### START: This is what you have to change

    # parse the input JSON dictionary
    input = json.loads(ciphertext)
```

```
# derive 2 keys from the password using Scrypt
kdf = Scrypt(
    salt=string2bytes(input['salt']),
    length=64,
    n=2**14,
    r=8,
    p=1
)
key = kdf.derive(password.encode('utf-8'))
key1 = key[0:32]
key2 = key[32:64]

# recompute the MAC tag and check
hm = hmac.HMAC(key2, hashes.SHA3_512())
hm.update(string2bytes(input['iv']))
hm.update(string2bytes(input['ciphertext']))
tag = hm.finalize()
if not(constant_time.bytes_eq(tag, string2bytes(input['tag']))):
    assert False, "MAC␣tag␣verification␣failed"

# decrypt the ciphertext
cipher = Cipher(algorithms.AES(key1), modes.CFB(string2bytes(input['iv'])))
decryptor = cipher.decryptor()
plaintext = decryptor.update(string2bytes(input['ciphertext'])) + decryptor.finalize()

### END: This is what you have to change

# decode the byte string back to a string
return plaintext.decode('utf-8')
```

(c) [1 mark] How many bytes of overhead does your encryption function add on top of the minimum number of bytes required to represent the message itself?

Solution. 64 bytes: 16 for the Scrypt salt, 16 for the AES-CFB IV, 32 for the MAC tag

(d) [1 mark] 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?

Solution. For $n = 2^{14}$, $r = 8$, $p = 1$, the estimated runtime is around 100 ms (measured to be about 0.078 seconds on the `jupyter.math.uwaterloo.ca` server).

According to `https://stackoverflow.com/questions/11126315/what-are-optimal-scrypt-work-factors/30308723#30308723`, the memory needed for a single Scrypt operation is

$$128 \text{ bytes} \times N \times r$$

which for the parameters above is 16 MB.

(e) [1 mark] What is one mistake that someone could make when solving this question that could undermine security?

Solution.

- Not use a random salt for Scrypt.
- Reuse the AES-CFB IV.
- Use the same key for encryption and MAC.
- Use MAC-then-encrypt or encrypt-and-MAC.
- Return the decrypted ciphertext even if the MAC check fails.
- Not use a cryptographically secure pseudorandom generator.

2. [8 marks] **Password hash cracking**

This question uses randomization to customize to you specifically. Please include your max-8-character UW user id (`b54khan`) at the beginning of your answer so we can look up your custom solution.

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.

(a) [2 marks] 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.

Solution. Alice's password is `429933` .

(b) [6 marks] On `example.org`, they have stricter security requirements. First, they prepend a 64-bit salt k_1 to each password. The same salt k_1 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 k_2. Again, the same key k_2 (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:

$$c = \mathsf{AES.Enc}(k_2, \mathsf{H}(k_1 \| pw))$$

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

i. [2 marks] Is it feasible to carry out a exhaustive search to recover the salt and encryption key? Why or why not?

Solution. No. An exhaustive search on the combined salt and key space would require testing $2^{64} \times 2^{64} = 2^{128}$ combinations, which is infeasible.

ii. [2 marks] 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.

Solution. We can use a meet-in-the-middle attack. First, we use our chosen plaintext oracle from part (a) to determine 2 plaintext/ciphertext pairs, (m_1, c_1) and (m_2, c_2) (with high probability, this should be enough to determine the unique key used). Then, we behave as follows:

```
for each k ∈ {0,1}^64 :
    Compute h_k = H(k||m_1)
    Store (h_k, k) in a table (sorted by first entry)
for each k' ∈ {0,1}^64 :
    Compute x = AES.Dec(k', c_1)
    if x = h_k for any entry (h_k, k) in the table :
        if AES.Enc(H(k||m_2)) = c_2 :
            return (k, k')
```

Notice that this algorithm is efficient: it takes about 2^{65} total hash/encryption operations, and since the table is sorted by first entry, table lookups can be done in logarithmic time.

iii. [1 mark] Would the same attack work if `example.org` first encrypted each password and then hashed it with their secret salt? Why or why not?
Solution. No, the same attack would not work, since cryptographic hash functions are preimage resistant.

iv. [1 mark] Would you recommend using this method of password hashing? Why or why not?
Solution. No. This method is deterministic, so anyone who gets their hands on the password database will be able to tell whether two users have the same password.

3. [4 marks] **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.

(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:

$\mathsf{BIGMAC}(k, m)$

1 : **for** $i = 1, \ldots, n$:
2 : $\quad t_i \leftarrow \mathcal{M}(k, m_i)$
3 : **return** $t_1 \| \ldots \| t_n$

Solution. This is not secure. To attack it, choose two distinct blocks m_1, m_2 and demand the tag (t_1, t_2) of (m_1, m_2). You can now forge the tag of (m_2, m_1) since it is (t_2, t_1).

(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:

$\mathsf{MACNCHEEZ}(k, m)$

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

Solution. This is not secure. To attack it, choose two distinct blocks m_1, m_2 and demand the tag $t_1 := \mathcal{M}(k, m_1)$ of m_1. Then ask for the tag $t_2 = \mathcal{M}(k, \mathcal{M}(k, m_1) \oplus m_2)$ of (m_1, m_2). You can now forge the tag of $t_1 \oplus m_2$ since it is t_2.

4. [6 marks] **Authenticated encryption using ChatGPT**

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.

[2] Good news: I am not out of a job yet.

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:

1. Choose a secure symmetric key encryption scheme for E_{k_E} and explain why it is secure.
2. Choose a secure MAC for M_{k_M} and explain why it is secure.
3. Prove that the proposed authenticated encryption scheme is not secure when using your encryption scheme and MAC by explaining how to attack it.

Added October 29: Clarification of what it means for an authenticated encryption scheme to be "secure": An authenticated encryption scheme should provide both confidentiality and integrity.

- For confidentiality, we can use the standard IND-CPA security definition that we have used previously: it should be hard for the adversary to learn any information about the plaintext given the ciphertext, even if they are given the powers of a chosen plaintext attack in which they can submit plaintexts and receive the corresponding ciphertexts.
- For integrity, we want to achieve something similar to the unforgeability under chosen message attack property for message authentication codes, but we have to phrase it slightly differently because we're working with an authenticated encryption scheme. Our integrity definition will be that it should be computationally infeasible for the adversary to generate a ciphertext that is successfully decrypted by the authenticated encryption scheme's decryption algorithm (i.e., does not "return error"), given the powers of a chosen plaintext attack in which they can submit plaintexts and receive the corresponding ciphertexts generated by the authenticated encryption scheme. Note that in order for this definition to not be trivial, the ciphertext that the adversary submits to win the game must be different from all of the ciphertexts it received using its chosen plaintext attack oracle. In other words, it has to create one more valid ciphertext. This is called INT-CTXT (integrity of ciphertexts).

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.

(a) "Cascading Encryption and MAC Structure":

$\mathsf{AEON}(k = (k_E, k_M), m)$

1 : $C \leftarrow E_{k_E}(m)$
2 : $T_1 \leftarrow M_{k_M}(m)$
3 : $T_2 \leftarrow M_{k_M}(T_1 \| C)$
4 : **return** (C, T_1, T_2)

Solution. Here's the decryption and verification algorithm:

$\mathsf{AEON.Dec}(k = (k_E, k_M), (C, T_1, T_2))$

1 : **if** $T_2 \neq M_{k_M}(T_1 \| C)$ **then return error**
2 : $m \leftarrow E^{-1}_{k_E}(C)$
3 : **if** $T_1 \neq M_{k_M}(m)$ **then return error**
4 : **return** m

Here's an instantiation of AEON that is insecure. Choose any secure encryption scheme. Choose any secure MAC that completely leaks its plaintext (you can always add the plaintext to the output of any secure MAC and keep it secure). T_1 leaks the plaintext and breaks confidentiality.

(b) "XOR-Dependent Combined Encryption and MAC":

$\mathsf{AETHER}(k = (k_E, k_M), m)$

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

Solution. Here's the decryption and verification algorithm:

$\mathsf{AETHER.Dec}(k = (k_E, k_M), (C, T, N))$

1 : **if** $T \neq M_{k_M}(C \oplus N)$ **then return error**
2 : $m \leftarrow E^{-1}_{k_E}(C) \oplus N$
3 : **return** m

Choose a secure encryption scheme that is a stream cipher, like AES in counter (CTR) mode. Choose any secure MAC. Given one valid ciphertext (C, T, N) for message m, then, for any value of W of the same length as N and m, $(C' = C \oplus W, T, N' = N \oplus W)$ is a encryption of plaintext $m \oplus N$. This is because, during MAC verification, we check if $T = M_{k_M}(C' \oplus N') = M_{k_M}(C \oplus W \oplus N \oplus W) = M_{k_M}(C \oplus N)$ which is true. The resulting plaintext is $E^{-1}_{k_E}(C') \oplus N' = E^{-1}_{k_E}(C \oplus W) \oplus N \oplus W = E^{-1}_{k_E}(C) \oplus W \oplus N \oplus W = E^{-1}_{k_E}(C) \oplus N = m \oplus N$, where the 2nd equality sign follows from the additive nature of stream ciphers.

5. [9 marks] **RSA encryption**

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,[citation needed] 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).

(a) [2 marks] Show that, if $m_1, m_2 \in [0, N-1]$ with RSA encryptions c_1 and c_2, respectively, then

$$\mathsf{RSA.Enc}((N,e), m_1 \cdot m_2) = c_1 \cdot c_2 \bmod N$$

Solution. We have that $c_1 \equiv m_1^e \bmod N$ and $c_2 \equiv m_2^e \bmod N$. So, we get that

$$(m_1 m_2)^e \equiv m_1^e m_2^e \equiv c_1 c_2 \bmod N$$

(b) [4 marks] Show how you can determine whether $m^* > \frac{N}{2}$ or $m^* < \frac{N}{2}$, by interacting with the RSA plaintext parity checking oracle. Justify mathematically why your approach works.

Solution. By sending the RSA plaintext parity checking oracle a ciphertext and seeing the plaintext response we receive back, we can determine whether the message corresponding to that ciphertext is even or odd.

Since N is an RSA modulus, it is the product of two odd primes, and is therefore odd.

Let $m' \in [0, N-1]$ be such that $2m^* \equiv m' \bmod N$.

Notice that if $m^* < \frac{N}{2}$, then $2m^* < N$ and hence $m' = 2m^*$ (with equality as integers, not being reduced modulo N). In particular, in this case we have that m' is even.

On the other hand, if $m^* > \frac{N}{2}$, then $N < 2m^* < 2N$ and hence $m' = 2m^* - N$ (as integers). In particular, in this case we have that m' is odd.

So, we can determine whether $m^* > \frac{N}{2}$ or $m^* < \frac{N}{2}$ by querying the oracle on $2^e c^*$ (by part (a), $2^e c^*$ is the RSA encryption of $2m^*$), and then concluding that $m^* < \frac{N}{2}$ if the oracle indicates that m' is even, or concluding $m^* > \frac{N}{2}$ if the oracle indicates that m' is odd.

(c) [3 marks] 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.

Solution. We can do a binary search for m^* over the message space using a similar technique to part (b). Let $\mathcal{O}$ be our even/odd oracle. Our attack is as follows:

1 : $L \leftarrow 0$
2 : $R \leftarrow 0$
3 : **for** $i = 1, \ldots, \lceil \log_2(N) \rceil$:
4 : $\quad \frac{a}{b} \leftarrow \frac{L+R}{2}$ (as a fraction in lowest terms)
5 : $\quad$ **else if** $\mathcal{O}((b)^e c^*) =$ "even" : $R \leftarrow \frac{a}{b}$
6 : $\quad$ **else** : $L \leftarrow \frac{a}{b}$
7 : **return** $\lfloor R \cdot N \rfloor$

Correctness: The outer framework of our algorithm is essentially binary search (although we keep our endpoints as fractions to make some math easier). In round 1, we determine whether $0 \le m^* < \frac{N}{2}$ or $\frac{N}{2} \le m^* < N$ in exactly the way described in part (b). In the i-th round (for $i \ge 1$), we have from the $(i-1)$-th round a value x_{i-1} such that $\frac{x_{i-1}}{2^{i-1}} \cdot N \le m^* < \frac{x_{i-1}+1}{2^{i-1}} \cdot N$. We then compare m^* to the middle value of that interval (that is, $\frac{2x_{i-1}+1}{2^i} \cdot N$), similarly to what we did in part (b). In our algorithm, this is $\frac{a}{b} \cdot N$, and $\frac{a}{b}$ is already in lowest terms since b is a power of 2 and a is odd. Let $m' \in [0, N-1]$ be such that $2^i m^* \equiv m' \bmod N$. If $\frac{x_{i-1}}{2^{i-1}} \cdot N \le m^* < \frac{2x_{i-1}+1}{2^i} \cdot N$, then $2x_{i-1}N \le 2^i m^* < (2x_{i-1}+1)N$. So, m' is obtained from $2^i m^*$ by subtracting N and even number of times, and hence in this case m' is even. Otherwise, if $\frac{2x_{i-1}+1}{2^i} \cdot N \le m^* < \frac{x_{i-1}+1}{2^{i-1}} \cdot N$, then $(2x_{i-1}+1)N \le m^* < (2x_{i-1}+2) \cdot N$. So, m' is obtained from $2^i m^*$ by subtracting N an odd number of times, and hence in this case m' is odd. This is precisely the information obtained by querying the oracle on line 5, in each case,

our algorithm updates the bounds on m^* correctly. Finally, after $\lceil \log_2(N) \rceil$ rounds, there will only be a single integer left between $L \cdot N$ and $R \cdot N$, which can be found by taking $\lfloor R \cdot N \rfloor$ (or, equivalently, by taking $\lceil L \cdot N \rceil$).

Efficiency: The algorithm makes $\lceil \log_2(N) \rceil$ (which is the bit length of N) rounds of interaction with the RSA plaintext parity checking oracle.

Academic integrity rules

You should make an effort to solve all the problems on your own. You are also welcome to collaborate on assignments with other students in this course. However, solutions must be written up by yourself. If you do collaborate, please acknowledge your collaborators in the write-up for each problem. *If you obtain a solution with help from a book, paper, a website, or any other source, please acknowledge your source. You are not permitted to solicit help from other online bulletin boards, chat groups, newsgroups, or solutions from previous offerings of the course.*

Due date

The assignment is due via Crowdmark by 11:59:59pm on November 4, 2024.

If you are requesting an extension or accommodation because of a verification of illness form, self-declared student absence, or AccessAbility Services accommodation, see the "Missed assignment / extension policy and forms" instructions on the CO 487/687 LEARN site.

Changelog

- Fri. Oct. 18: assignment posted
- Tue. Oct. 29: added clarification in question 4