- The inner padding constant is the byte 0x36 repeated (Block size) times, the outer padding constant is the byte 0x5c repeated (Block size) times.
- In grading at our side, the message, shared key, and lengths will be chosen at random and everything must work.
- Compile your code with the following commands:
  gcc hmac.c -lssl -lcrypto -o hmac
- Run your code for the first test files with the following commands:
  ./hmac Message1.txt SharedKey1.txt
- We have provided a script (VerifyingHMAC.sh) and test files that you can use to test the correctness of your final code (hmac.c). Please, only submit your code if it is functional.
- You can check your answer with each provided test file manually or you can use the script to verify your solution for the HMAC programming assignment.
- In order to use the script, just put "hmac.c" and the rest of the provided files along with the "VerifyingHMAC.sh" in one folder and run the following command in the terminal:
  bash VerifyingHMAC.sh
- The verification script is the same one that is run to grade your submissions, only the test vectors are different.
- **You Must submit 1 file for this part of the assignment:**
  hmac.c

Rubric:

- The **"Key.txt"** file **[10 pt]**
- The **"ProcessedKey.txt"** file **[10 pt]**
- The **"FinalHash.txt"** file **[10 pt]**

3 Programming Assignment 2: Challenge-Response Protocol [45 pt]

In this exercise we will put into action a Challenge-Response protocol similar to the ones that were discussed during the class. Specifically, consider how storing and using a counter and nonce as state information helps secure the authenticity and integrity of the message as described in "XX.ppt", Slide X.

In this toy Challenge-Response implementation, Alice will have a message and shared secret key, as well as storing an incrementing counter and nonce as state information. Alice will encrypt the message by XORing it with the Hash of the concatenation of the secret key and counter, then sign the encrypted message (ciphertext) by using a keyed HMAC on the concatenation of the ciphertext and the nonce. Alice will send the ciphertext and signature to Bob as her Challenge. Bob, at this point, has the ciphertext, signature, shared secret key, and stores the same counter and nonce as Alice as his state information. Bob checks Alice's signature to see if it matches, as he has the shared secret key, ciphertext, and the same nonce state as Alice. If the comparison is successful, Bob continues by decrypting the ciphertext with XOR in much the same manner as it was encrypted, getting the message back in plaintext. Bob computes his Response using the decrypted message, his counter (incremented by 1), and his nonce (incremented by 1). Bob sends this Response back to Alice, and increments his counter and nonce states for the next challenge/response with Alice. Alice receives Bob's Response and compares this with the expected response, which is calculated the same as Bob's, just with Alice's counter and nonce. If the comparison is successful, Alice can be confident that Bob has received the accurate message, and Alice increments her counter and nonce states for the next challenge/response with Bob.

We will utilize SHA256 from the OpenSSL libraries as our hashing function and utilize the HMAC function from OpenSSL to get our keyed HMAC results.

Alice:

1. Alice reads the message, shared secret key, counter state, and nonce state from **"Message.txt"**, **"SharedKey.txt"**, **"A_ctr.txt"**, and **"A_nonce.txt"** respectively.
   (Read them as unsigned char so that you can use the functions provided in in-class exercises).
2. Alice writes the key in Hex format to a file named **"Key.txt"**.
3. Alice encrypts the message with XOR to attain the ciphertext: $c \leftarrow m \oplus H(k||ctr)$.
4. Alice writes the ciphertext in Hex format to a file named **"Ciphertext.txt"**.
5. Alice computes the signature using a keyed HMAC: $(sig \leftarrow HMAC_k(c||nonce))$.
6. Alice writes the signature in Hex format to a file named **"Signature.txt"**.
7. Once Bob has processed the message, Alice reads Bob's response from the file named **"Response.txt"**.
8. Alice computes what the correct response should be from Bob $(response' \leftarrow H(m||(ctr+1)||(nonce+1)))$ and compares it with Bob's response (Note that Bob wrote the response in Hex format).
9. If the comparison is successful, Alice writes "Acknowledgment Successful" in a file called **"Acknowledgment.txt."** Conversely, if the comparison fails, she records "Acknowledgment Failed.".
10. Alice increments and updates her counter and nonce files **"A_ctr.txt"** and **"A_nonce.txt"**.

Bob:

1. Bob reads the ciphertext, signature, shared secret key, his counter state, and his nonce state from **"Ciphertext.txt"**, **"Signature.txt"**, **"SharedKey.txt"**, **"B_ctr.txt"**, and **"B_nonce.txt"** respectively.
   (Read them as unsigned char so that you can use the functions provided in in-class exercises).
2. Bob computes the expected signature using a keyed HMAC in the same manner as Alice $((sig' \leftarrow HMAC_k(c||nonce)))$ and checks it against the signature that Alice provided (Note that Alice wrote the signature in Hex format).
3. If the comparison is successful, Bob continues, otherwise the program exits.
4. Bob decrypts the ciphertext with XOR: $m \leftarrow c \oplus H(k||ctr)$.
5. Bob computes his response: $response \leftarrow H(m||(ctr+1)||(nonce+1))$.
6. Bob writes his response in Hex format to a file named **"Response.txt"**.
7. Bob increments and updates his counter and nonce files **"B_ctr.txt"** and **"B_nonce.txt"**.

Programming Assignment 2 - Specific Notes

- The Hash (SHA-256) and HMAC functions are called from the OpenSSL libraries. You can refer to the OpenSSL library documentation https://www.openssl.org/docs/manmaster/man3/ to learn how to use these functions.
- The counter and nonce stored in the text files are decimal strings depicting integers in unsigned char form, it is intended to concatenate them as unsigned char but increment them as integers (so a nonce of the string "99" in unsigned char format is incremented to the string "100" in unsigned char format).

- Since Alice needs Bob's response to continue, if Bob's response does not exist yet then Alice must exit gracefully until the program is called again after Bob is run.
- In grading at our side, the seed, message, counter, nonce, and lengths will be chosen at will and everything must work. The message length, however, is guaranteed to be 256 bits or 32 bytes.
- Compile your codes with the following commands:
  gcc alice.c -lssl -lcrypto -o alice
  gcc bob.c -lssl -lcrypto -o bob
- Run your codes for the first test files with the following commands:
  ./alice Message1.txt SharedKey1.txt A_ctr.txt A_nonce.txt
  ./bob Ciphertext.txt Signature.txt SharedKey1.txt B_ctr.txt B_nonce.txt
- We have provided a script (VerifyingAliceBob.sh) and test files that you can use to test the correctness of your final codes (alice.c and bob.c). Please, only submit your code if it is functional.
- You can check your answer with each provided test file manually or you can use the script to verify your solution for this programming assignment.
- In order to use the script, just put "alice.c", "bob.c", and the rest of the provided files along with the "VerifyingAliceBob.sh" in one folder and run the following command in the terminal:
  bash VerifyingAliceBob.sh
- The verification script is the same one that is run to grade your submissions, only the test vectors are different. Note how the verification script runs the programs in order: first Alice, then Bob, then Alice.
- **You Must submit 2 files for this part of the assignment:**
  alice.c & bob.c

Common Assignment Notes

- For the correctness of reading from a file and XORing, please use unsigned char.
- For your convenience, we have provided a template file with necessary functions and descriptions: Required FunctionsHW1.c
- You may use the functions in RequiredFunctionsHW1.c, modify them, or write your own, modifying the RequiredFunctionsHW1.c file as necessary. **If you modify it, submit your modified RequiredFunctionsHW1.c file alongside your other code files in Canvas.**
- The code that throws errors or does not successfully compile/run on our side, receives zero credit (It doesn't matter if it works on your computer!).
- Follow the Instructions and all the specific names of output files mentioned above, otherwise there will be 20% deduction for inconvenience. If you followed the above steps and the output files are correct, you will receive full credit.

Rubric:

- The **"Key.txt"** file **[5 pt]**
- The **"Ciphertext.txt"** file **[10 pt]**
- The **"Signature.txt"** file **[10 pt]**
- The **"Response.txt"** file **[10 pt]**
- The **"Acknowledgment.txt"** file **[5 pt]**
- The **"A_ctr.txt"**, **"A_nonce.txt"**, **"B_ctr.txt"**, **"B_nonce.txt"** files **[5 pt]**