Swiss Army Cryptographic Toolset for beginners

A user manual and guide for cryptographic tools

Moaz Khan
Rushi Patel
Jeet Desai
Yash Patel

Table of Contents

Swiss Army Cryptographic Toolset User Manual

The cryptographic tasks that we decided to include in the toolset are:

- Symmetric block cipher encryption/decryption (3DES, AES)
- Asymmetric key encryption and signature verification (RSA)
- Message Authentication (HMAC)
- AES Key Wrapper

Triple Data Encryption Algorithm (3DES)

- In order to use 3DES enter "1" on the main menu.
- You will then be asked to enter a message to encrypt. **Remember that the message size has be in multiples of 8 characters (8 bytes).**
- After entering the message in appropriate length, 3DES will compute the encryption and provide you a ciphertext of your message.
- After encryption, an explain of the encryption process will be provided.

```
Decrypted Text:  b'secret12'

Decryption Process:
1) Decryption of a ciphertext is in reverse process.
2) User first decrypt using K3, then encrypt with K2, and finally decrypt with K1.
```

- The decrypted message will now be shown and its process.
- Finally, enter "yes" or "no" when prompted to see the parameters that were used in the encryption/decryption process of 3DES

```
Would you like to know the parameters that were used in encryption/decryption process? yes or no

yes
```

Advanced Encryption Standard (AES)

- In order to use AES enter "2" on the main menu.
- You will then be asked to enter a message to encrypt. **Remember that the message size has be in multiples of 16 characters (16 bytes).**
- After entering the message in appropriate length, AES will compute the encryption and provide you a ciphertext of your message.

```
Note: Input strings must be a multiple of 16 characters in length!
Enter a message you would like to encrypt:

secretmessage123

Message length is appropriate

Encrypted Text:  b'W0\xb0\x8c\xc2L\xba\xc8\x12\x83\xf1\xa6EG?\x83'
```

- After encryption, an explain of the encryption process will be provided.
- The decrypted message will now be shown and its process.

```
Decrypted Text:  b'secretmessage123'
```

- Finally, enter “yes” or “no” when prompted to see the parameters that were used in the encryption/decryption process of AES.

```
Would you like to know the parameters that were used in encryption/decryption process? yes or no

yes
```

Rivest Shamir Adleman (RSA) Algorithm:

Sample run of RSA:

- **Step 1**: get the public and private keys
- Enter the message you wish to send = **“hello world”**
- n = 2048 and e = 257
- To display more information when prompted, click ‘y’ for yes and ‘n’ for no

```
Please Enter the plaintext 'M' Alice wishes to send: hello world
---> Generating Alice's private and public keys...
---> Generating Bob's private and public keys...


do you wish to know how to generate  the n value?: 'y' or 'n' ?y
the n value(RSA Modulus) is the product of 2 prime numbers
---> n = p*q =  2048

do you wish to know how to generate the e value?: 'y' or 'n' ?y
e must be smaller than the toitent of n and is coprime to n
--->e=  257
```

- **Step 2**: encryption

```
generating cipher text...

the function used to encrypt = C = (M^e % n)

The Ciphertext 'C' that Bob receieves is  b"\rr6\xb3\x10\r\xeaM\xdd?\xf7\xfa\xb2\x8e\xd4\xfbr\x9cg
\x8e\xf3'\xa8B\x1a\xa4\xe0\x0f\x88.\x88\x0f\xcc\xae\xc4\x96\xb2\xdc2\xbe\xc0U\xc3\n\x8b\xf2\xbb
\xaf\x1b=Y\xc6\x13\xc8\xab\x1e\x06(\xcd5\xaaT$\x0fJ&\r7\x98\x97h\xad8v=\x02O\xb8\xee\x9bb\x0em
\xe1\xef\x17\x8eDM\x1fN\xaf\xa9\x99\xeamN\x00\x83\xa2\xc0\xe4)?\xfc%\x9a\x90&
\xe1\x164\x90\xe2\xf3\x19\x84n\xc3\x91b\x04\xa5'Q\x11\x02\x86\xdf\x17\xdfU=\xef\x17$fq\xdd\xda\xed
\xea^_\xbd\xa9m\x00\r\x1c\xdfqK\xb0\x1b\xb8\xdd\xfc\xd2\x05\xef\xd0<\xa0\x8d\xb4\xca\x92\xf9P\xbf
\xa2F\xff\xc9)uOBh`=\xf5\xd2\xb5Z#ae5\xc5Y\xaf\xfer\xfd\x13\xc8J\xd6FgD\x86\xfaC\xc2\xac\xc7\x9a
\x06\xdcT\x1fj\xd8\x15\x8e\xc9\xd1\xa1\x11\x97\x07W\xc9B\xb0\xef\xd5\x1a\x04\x9f\x05Q\x1a\xc4GX
\xb9Q\xa8\x0e\xb9\xa8\xf3,\xfc\xc1n\x8byr\xb3\xa2\x1f"
```

```
would you like to know about signing process:'y' or 'n'? y
 in RSA, a private key is used to sign a message
---> this allows anyone with the public key to verify
that the message was from the correct private key holder
Alice signed the message with her signature
---> this signature had a hash of SHA512
---> the padding OAEP is used to get a desired length of output
---> regardless of the input used for bytesize
```

- **Step 3**: decrypt

```
Do you wish to see the signature? 'y' or 'n'? y
The Signature used for the RSA was:  b'!\xc8_\xe3\x85\x02?\\\x91"dl\x8a\xff1\xf6\x9b\x8f\xd9\xc5V
\xd9U\xf8\xa8A\x13-\x07z\xd7zy\xefCxga,e\xf9%\x9a\x97\x01\xc7AT\xa8D:\x85>@\xa7r%\x0f\xa9\xb4\xcb
\xc9</\xe6\xad\xa5\x82D\xea\xed\xaeS\xec\xafX\xb1\x1fA\xe2\xcc\x1d\x1e\x7f&!:["\x94\x03\xe3\xd2Q
\xee{\x15\xfff\xb8@\t\xe0\xa2_\x03\xa7\xafQ`\xe5\x8c\x13\xf1\xa4b\xe2\xdb\x9c
\x94\xf2\x04\xe1\xa15\xec\xe3z`?QTT\x95\x80v\xab=_\xb8\x11\x9a\x93\x99\x14\n\x88\x8dgf\x10a
\xf9\x80\xd5\x17\x8f\xf5\xac\xdf\xa1\xe1wI\'3\x90\xdbi\xb9&\xc9\xd6\x9a\xe9G\x1a~6\xa3Q\xd0\xc9u!
\xb3\xc2\xc4\x060\x9f\xdd\xc2(\xb2\xc4qUk.\xd9\xd48\xb6\x7f\x1f\xd8\xeda\x11K\xdb\x8e
\xd2\xa3\xa7\xaa\x83\xfc!\xf3\xe4\xcdUYs|\xcf\xb1\xab\xf6\xc4*YW\xec\xb2\xc8\xc2p@\xde\xfb\x0c\xbe
\xf4.\x9a\x19\xcd\xe7W1\n\t\xf0'


Bob encrypts the message, verifys the signature successfully and securely gets the original plain
text:  b'hello world'
==================================
Exiting RSA_ENCRYPTION tutorial...
```

Hash Message Authentication Code:

A Sender hashes a message with the private key to generate a hash code using the SHA256 algorithm.

SENDER's Perspective:

Step 1: Enter 1 if the user is a sender and 2 if the user is a receiver
Step 2: Enter a private key which is only shared between a sender and a receiver. Make sure this private key is 32 characters long
Step 3: Enter a message which the user wants to hash
Step 4: A hash code of the message and the private key will be printed

```
----------------------------------------Hash Message Authenticatio Code (HMAC)--------------------------------

Are you a sender(1) or receiver(2)?
1

Enter a key whose length is 32 characters: 12345678901234567890123456789012

Enter a message to hash:This is the original message

Your message: b'This is the original message'

Your hash is bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9
```

Printing an output explanation for the whole process which took place at the sender's side.

```
--------------------------------------Sender's Process--------------------------------------

What just happened?

Sender has a message which he/she wants to send it to someone

But what is the guarantee that this message will not be altered on it's way to the receiver

How would receiver know that this message was not changed or altered on its way?

This problem questions the authenticity of the message

To prevent this problem sender will use that message and a private key which is only shared between him and the receiver

Sender will take this message and hash it with the private Key using SHA256 algorithm

Sender will then append the original message with the hash created using the private key and send it to the receiver

----------------------------------------END-----------------------------------------------------
```

A receiver is trying to hash the message with a shared private key using SHA256. Once the hashing is complete receiver will then, compare the received hash code with the hash code which receiver generated using the message. As you can see here both hashes are same which means this message is authentic.

RECEIVER Perspective:

Step 1: Enter 1 if the user is a sender or 2 if the user is a receiver
Step 2: Enter a private key which is only shared between a sender and a receiver. Make sure this private key is 32 characters long
Step 3: Enter a message which was sent by the sender
Step 4: Enter a hash code which was received with the message
Step 5: Sender's and receiver's hash is compared in this step and in this case, both the hashes are matching (Look at the very bottom)

```
----------------------------------------Hash Message Authenticatio Code (HMAC)--------------------------------

Are you a sender(1) or receiver(2)?
2

Enter a key which is shared between sender and yourself (Must be exactly 32 characters long):
12345678901234567890123456789012

Enter a message to hash: This is the original message

Enter the hash code you received with the message: bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9

Your hash code and sender's hash matches!

Sender's Hash:  bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9
Your hash:  bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9
```

This is the same process as the above picture but the only difference is that hashes do not match in this process. At the very bottom, you can see that both hashes are very different from each other

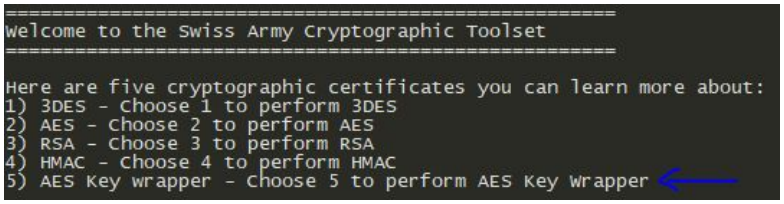
```
----------------------------------------Hash Message Authenticatio Code (HMAC)--------------------------------

Are you a sender(1) or receiver(2)?
2

Enter a key which is shared between sender and yourself (Must be exactly 32 characters long):
12345678901234567890123456789012

Enter a message to hash: This is not the original message

Enter the hash code you received with the message: bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9

Hashes do not match

Please make sure you are using the shared private key which is shared between you and the sender
If you are using the same private then the message or the hash have been altered

Sender's Hash:  bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9
Your hash:  cbc170f423fdf551510dd6358df7733740ca35d2d4960d03cb806ecd391a5427
```

Printing an output which explains the whole process on the receiver's side.

```
---------------------------------------Receiver's Process---------------------------------------

What just happened?

Receiver receives a message from the sender with a hash attached to it

But what is the guarantee that this message was not be altered on it's way to the receiver

How would receiver know that this message was unchanged or altered on its way?

This problem questions the authenticity of the message

To prevent this problem receiver will use that message and a private key which is only shared between him and the sender

Receiver will take this message and hash it with the private Key using SHA256 algorithm

Receiver will then compare the hash he/she received with the message sent my sender

---------------------------------------END-------------------------------------------------------
```

Key Wrap (KW) Mode of Operation

In order to use the Key Wrap Mode in the toolset, select option 5 on the main menu.



Next, when prompted, enter "y" if you would like to set your own value for the key encryption key (wrapping key).
If you entered "n", the key encryption key value will default to b'1234567890123456'.

```
For simplicity's sake, we have already defined a value for the key encryption key to be b'1234567890123456'.
Do you wish to change this value?(y/n):n
```

Similarly, after setting the key encryption key value, you will be prompted to set your own value for the new symmetric key which is to be wrapped (key to wrap). Enter “y” to do so.
If you entered “n”, the value will default to b’wow-this-is-fun!’.

```
Do you wish to change this value?(y/n):y
Enter a new value for key to be wrapped.(MUST BE OF 16, 24 OR 32 BYTES):12345678912345678912345
```

KEEP IN MIND BOTH VALUES MUST BE 16, 24 OR 32 BYTES!!!!!

After the program computes the wrapped key value using AES encryption, you will be prompted to press enter to see how the value was computed.
Press enter again to see how the un-wrapping process works.

```
Press Enter to see how we computed the wrapped key value...

What just happened?
-----------------------------------

So to go over how we got the encrypted value, lets take a mo
```

You are now a professional in utilizing our cryptographic toolset. Thank you.

Appendix

```
# -*- coding: utf-8 -*-
"""

Swiss Army Cryptographic Toolset
Rushi Patel (100615230)
Yash Patel (100621177)
Jeet Desai (100635399)
Moaz Khan (100593258)

"""
```

```
from Crypto import Random
from Crypto.Cipher import DES3
from Crypto.Cipher import AES
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import rsa
import base64
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
from Crypto.Hash import HMAC
from Crypto.Hash import SHA256
from cryptography.hazmat.primitives import keywrap

def DES3CIPHER():
    #ask user for message to encrypt
    print("\n -- 3DES -- \nIt is a symmetric-key block cipher. It performs encryption and
decryption"
          " of fixed data sized blocks (multiple of 8 characters (8 bytes).",
          " 3DES has the same functionality as DES, however 3DES applies the DES cipher 3
times")
    print("\nNote: Input strings must be a multiple of 8 characters in length!")
    print("Enter a message you would like to encrypt: ")
    message = input() #store user inpunt in a string variable

    #checking whether the length of user input is multiple of 8.
    #To check if its multiple of 8, taking the modulo of the input length by 8.
    #If the remainder is 0, do encryption/decryption of user's message.
    #If remainder is not 0, continue asking the user for input in appropriate message length
    while len(message) % 8 != 0:
        print("\nInput strings must be a multiple of 8 characters in length!")
        print("Please enter your message again: ")
        message = input()

    #define key in 16 bytes
    key = 'Sixteen byte key'

    #define initialization vector
    iv = Random.new().read(DES3.block_size) #DES3.block_size==8

    #encrypt message
```

```
    cipher_encrypt = DES3.new(key, DES3.MODE_OFB, iv)
    encrypted_text = cipher_encrypt.encrypt(message)


    print("\nMessage length is appropriate")

    print("\nEncrypted Text: ", encrypted_text)
    print("\nEncryption Process: ")
    print("Triple DES takes 3 keys for encryption. Each key is 64 bits long,"
          " however 8 bits from the key are striped and saved for redundancy."
          " After taking out the bits, the key becomes 168 bits."
          " The key from the user is broken down into three keys and padded if the bits"
          " don't make up to a valid key size.")
    print("1) First encrypt the plaintext blocks using single DES with key K1.")
    print("2) Now decrypt the output of step 1 using single DES with key K2.")
    print("3) Finally, encrypt the output of step 2 using single DES with key K3.")
    print("4) The output of step 3 is the ciphertext.")

    #Decrypt message
    cipher_decrypt = DES3.new(key, DES3.MODE_OFB, iv)
    Decrypted_text = cipher_decrypt.decrypt(encrypted_text)

    print("\nDecrypted Text: ", Decrypted_text)
    print("\nDecryption Process: ")
    print("1) Decryption of a ciphertext is in reverse process.")
    print("2) User first decrypt using K3, then encrypt with K2, and finally decrypt with K1.")

    print("\nWould you like to know the parameters that were used in encryption/decryption
process? yes or no")
    decision = input()

    if (decision == "yes"):
        print("\nKey: ", key)
        print("\nRandomly Generated IV: ", iv)
        print("IV is used as an initial n-bit input block for the Ouput Feedback Mode")
        print("\nMode of Operation used: Output Feedback Mode (OFB)")
        print("The mode works as such, initially it uses an IV as the input for encryption,"
              " the output from the encryption function is used again in the next round as"
              " input for encrypting the next block of plaintext.")
    elif (decision == "no"):
```

```
        exit

def AESCIPHER():
    #ask user for message to encrypt
    print("\n -- AES Chosen -- \nAES is a symmetric encryption algorithm. It’s operation works as such,"
          " replacing inputs and shuffling bits around."
          " AES performs its computations based on rounds. The rounds in AES depends on the length of the key."
          "\n(10 rounds - 128-bit keys, 12 rounds - 192-bit keys, 14 rounds - 256-bit keys)"
          " Each round uses a different 128-bit round key.")
    print("\nNote: Input strings must be a multiple of 16 characters in length!")
    print("Enter a message you would like to encrypt: ")
    message = input() #store user inpunt in a string variable

    #checking whether the length of user input is multiple of 16.
    #To check if its multiple of 8, taking the modulo of the input length by 16.
    #If the remainder is 0, do encryption/decryption of user's message.
    #If remainder is not 0, continue asking the user for input in appropriate message length
    while len(message) % 16 != 0:
        print("\nInput strings must be a multiple of 16 characters in length!")
        print("Please enter your message again: ")
        message = input()

    #define key in 16 bytes
    key = 'Sixteen byte key'

    #define initialization vector
    iv = Random.new().read(AES.block_size)

    #encrypt message
    cipher_encrypt = AES.new(key, AES.MODE_CFB, iv)
    encrypted_text = cipher_encrypt.encrypt(message)


    print("\nMessage length is appropriate")

    print("\nEncrypted Text: ", encrypted_text)
    print("\nEncryption Process: ")
    print("Each round has 4 sub processes: ")
```

```
    print("1) Byte substitution - 16 byte input is substituted with values from a fixed table
(S-box)."
          " The result is stored in a 4 by 4 matrix")
    print("2) Shift Rows - each of the four rows from the byte substitution is shifted to the left."
        " Any entries from the matrix that fall off are inserted again to the right side of the row. ")
    print("   First row is not shifted")
    print("   Second row is shifted to the left by one position")
    print("   Third row is shifted to the left by two positions")
    print("   Fourth row is shifted to the left by three positions")
    print("3) Mix Columns - each column of four bytes are transformed using a mathematical
function."
        " This takes a column as one input and outputs a different different column with different
bytes."
        " At the end it results in a new 4 by 4 matrix. This step is not done in the last round")
    print("4) Add round key - the 16 bytes of the matrix are considered as 128 bits and are XORed
to"
        " the 128 bits of the round key."
        " The resulted 128 bits are used as 16 bytes in the next round."
        " In the last round the output is the cipher text.")


    #Decrypt message
    cipher_decrypt = AES.new(key, AES.MODE_CFB, iv)
    Decrypted_text = cipher_decrypt.decrypt(encrypted_text)

    print("\nDecrypted Text: ", Decrypted_text)
    print("\nDecryption Process: ")
    print("Each round has four sub processes. Does decryption in reverse order of encryption
process")
    print("1) Add round key")
    print("2) Mix columns")
    print("3) Shift rows")
    print("4) Byte substitution")

    print("\nWould you like to know the parameters that were used in encryption/decryption
process? yes or no")
    decision = input()

    if (decision == "yes"):
        print("\nKey: ", key)
```

```
        print("\nRandomly Generated IV: ", iv)
        print("IV is used as an initial n-bit input block for the Cipher Feedback Mode (CFB)")
        print("\nMode of Operation used: Cipher Feedback Mode (CFB)")
        print("The mode works as such, initially an IV is used for encrypting a block of plaintext,"
              " then the output is XORed with the plaintext. The result of the XOR is fed into the
next"
              " round as the 'IV' for encrypting the next block of plaintext.")
        print("")
    elif (decision == "no"):
        exit

#-----------------------------------------------------------RSA-------------------------------------

def RSA():

    print("==========================")
    print("Welcome to RSA encryption")
    print("==========================")
    print(" ")
    print("Key_terms: ")
    print("---> public exponent = e ")
    print("---> RSA modulus = n ")
    print("---> private exponent = d ")
    print("---> plain text = M ")
    print("---> cipher text = C ")
    print(" ")
    print(" ")
    e = 257
    n = 2048

    plain_text = str(input ("Please Enter the plaintext 'M' Alice wishes to send: " ))

    print("---> Generating Alice's private and public keys... ")
    print("---> Generating Bob's private and public keys... ")
    alice_private_key = rsa.generate_private_key(e,n,backend=default_backend())
    alice_public_key = alice_private_key.public_key()
    print(" ")
    selection1 = input("do you wish to know how to generate  the n value?: 'y' or 'n' ?")
    if(selection1 =='y'):
        print("the n value(RSA Modulus) is the product of 2 prime numbers")
```

```
        print("---> n = p*q = ",n)
    elif(selection1=='n'):
            print (" ")
    else: print("invalid operation!")

    selection2 = input("do you wish to know how to generate the e value?: 'y' or 'n' ?")
    if(selection2 =='y'):
        print("e must be smaller than the toitent of n and is coprime to n")
        print("--->e= ", e)
    elif(selection2=='n'):
            print (" ")
    else: print("invalid operation!")


    bob_private_key = rsa.generate_private_key(e,n,backend=default_backend())
    bob_public_key = alice_private_key.public_key()

    #Encodes the plain text
    plain_text_encoded= (plain_text).encode("utf-8")


#gets the ciphertext with a padding using mask generation function (MGF)
#for any given input, the desired output length will be the same
    cipher_text = alice_public_key.encrypt(plain_text_encoded,padding.OAEP

(mgf=padding.MGF1(algorithm=hashes.SHA512()),algorithm=hashes.SHA512(),label=None))

    print("generating cipher text...")
    print(" ")
    print("the function used to encrypt = C = (M^e % n)")
    print(" ")
    print ("The Ciphertext 'C' that Bob receieves is " , cipher_text)
    selection3 = input ("would you like to know about signing process:'y' or 'n'? ")
    if (selection3== 'y'):
        print(" in RSA, a private key is used to sign a message")
        print("---> this allows anyone with the public key to verify ")
        print("that the message was from the correct private key holder ")
        print("Alice signed the message with her signature")
        print("---> this signature had a hash of SHA512")
        print("---> the padding OAEP is used to get a desired length of output ")
```

```
        print("---> regardless of the input used for bytesize  ")
    elif(selection3=='n'):
        print(" ")
    else:print("invalid operation!")

    print (" " )
#decodes the cipher text from its base 64
    decoded_cipher_text = base64.b64decode(
    cipher_text) if not isinstance(cipher_text, bytes) else cipher_text
    plain_text2 =
alice_private_key.decrypt(decoded_cipher_text,padding.OAEP(mgf=padding.MGF1(algorithm=
hashes.SHA512()),algorithm=hashes.SHA512(),label=None)
        )
# signs the plain text in order for the correct user to receieve the information using the same hash
type and padding
    signature_RSA = alice_private_key.sign(
            plain_text_encoded,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA512()),
                salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA512()
                )

# Bob has to verify that the message is actually from Alice. only he can decrypt the message and
verify the signature
    bob_public_key.verify(
    signature_RSA,
    plain_text_encoded,
    padding.PSS(
            mgf=padding.MGF1(hashes.SHA512()),
            salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA512()
            )
    selection4= input("Do you wish to see the signature? 'y' or 'n'? ")
    if(selection4 == 'y'):
        print("The Signature used for the RSA was: ", signature_RSA)
    elif(selection4=='n'):
        print (" ")
```

```
    else: print("invalid operation!")

    encrypted_message_receieved = plain_text2
    print(" ")
    print(" ")
    print("Bob encrypts the message, verifys the signature successfully and securely gets the
original plain text: ",encrypted_message_receieved )
    print("====================================")
    print("Exiting RSA_ENCRYPTION tutorial...")

#-----------------------------------------------HMAC--------------------------------------

#12345678901234567890123456789012
#bce392c515c14a62106335639038deebd22e757504edbd2756924c5515c869a9

#Function which creates hash code of the message with the private key using SHA256 algorithm
def sender(key, message):
    h = HMAC.new(key, message, digestmod=SHA256)
    finalHash = h.hexdigest()
    print("\nYour message:", message)
    print("\nYour hash is", finalHash)

#Function which creates hash code of the message with the private key using SHA256 algorithm
#This function also checks whether hashes match or not
def receiver(key,message, hashString):
    h1 = HMAC.new(key, message, digestmod=SHA256)
    finalHash = h1.hexdigest()
    compare = hashString
    if(compare == finalHash):
        print("\nYour hash code and sender's hash matches! \n")
        print("Sender's Hash: " , compare)
        print("Your hash: " , finalHash)

    else:
        print("\nHashes do not match\n")
        print("Please make sure you are using the shared private key which is shared between you
and the sender")
        print("If you are using the same private then the message or the hash have been altered\n")
        print("Sender's Hash: " , compare)
        print("Your hash: " , finalHash)
```

```
def HMAC1():
    print("----------------Hash Message Authentication Code (HMAC)----------------\n")
    #Asking user whether they are a sender or a receiver
    ask = input("Are you a sender(1) or receiver(2)? \n")

    while((ask != '1') & (ask != '2')):
        ask = input("Please enter 1 for sender or 2 for receiver \n")

    #If user is a sender then call the sender function which does hashing
    if(ask == '1'):

        #Asking for a private key
        userInput = input("\nEnter a key whose length is 32 characters: ")

        #If the key is not 32 characters long then ask the user to enter a key which is 32 characters
long
        while((len(userInput)  !=  32)):
            userInput = input("\nEnter a key whose length is 32 characters: ")

        #Convert the user input into a string
        string = str(userInput)

        #converting strings to bytes
        key = string.encode('utf-8')

        #Asking user to enter a message to hash
        messageInput = input("\nEnter a message to hash:")

        #Convert the user input into a string
        words = str(messageInput)

        #converting strings to bytes
        message = words.encode('utf-8')

        #Calling the function of sender
        sender(key, message)
```

```
        #Explaining the process which takes place on the sender's side
        print("\n--------------------Sender's Process-------------------------")
        print("\nWhat just happened?")
        print("\nSender has a message which he/she wants to send it to someone")
        print("\nBut what is the guarantee that this message will not be altered on it's way to the
receiver")
        print("\nHow would receiver know that this message was not changed or altered on its
way?")
        print("\nThis problem questions the authenticity of the message")
        print("\nTo prevent this problem sender will use that message and a private key which is
only shared between him and the receiver")
        print("\nSender will take this message and hash it with the private Key using SHA256
algorithm")
        print("\nSender will then append the original message with the hash created using the
private key and send it to the receiver")
        print("\n--------------------END-----------------------")

    #If user is a sender then call the sender function which does hashing
    if(ask == '2'):

        #Asking for a private key
        userInput = input("\nEnter a key which is shared between sender and yourself (Must be
exactly 32 characters long): \n")

        #If the key is not 32 characters long then ask the user to enter a key which is 32 characters
long
        while(len(userInput) != 32):
            userInput = input("\n Enter a key whose length is 32 characters: ")

        #Convert the user input into a string
        string = str(userInput)

        #converting strings to bytes
        key = string.encode('utf-8')

        #Asking user to enter the message they received
        messageInput = input("\nEnter a message to hash: ")

        #Convert the user input into a string
        words = str(messageInput)
```

```
        #converting strings to bytes
        message = words.encode('utf-8')

        #Asking receiver to enter the hash code they received with the message
        hashInput = input("\nEnter the hash code you received with the message: ")

        #converting the hash code into a string
        hashString = str(hashInput)

        #Calling a function to check the integrity of the message
        receiver(key,message, hashString)

        #Explaining the process which takes place on the receiver's end
        print("\n------------------------Receiver's Process-----------------------")
        print("\nWhat just happened?")
        print("\nReceiver receives a message from the sender with a hash attached to it")
        print("\nBut what is the guarantee that this message was not be altered on it's way to the
receiver")
        print("\nHow would receiver know that this message was unchanged or altered on its
way?")
        print("\nThis problem questions the authenticity of the message")
        print("\nTo prevent this problem receiver will use that message and a private key which is
only shared between him and the sender")
        print("\Receiver will take this message and hash it with the private Key using SHA256
algorithm")
        print("\nReceiver will then compare the hash he/she received with the message sent my
sender ")
        print("\n--------------------------END-----------------------------")

def AESKeyWrap():
    print("\n\n\n==========================================")
    print("Welcome to Key Wrap (KW) Mode of Operation")
    print("==========================================")
    print(" ")

    print("Key wrapping and un-wrapping requires 3 parameters in order to function correctly:\n")
    print("\n1) The key encrytion key or 'the wrapping key', which must be of 16, 24 or 32 bytes in
value in order for AES to function.")
```

```
    print("2) The key that is to be wrapped 'key to wrap', this is also of 16, 24 or 32 bytes in value.")
    print("3) Finally, the wrapped key, this is the encrypted value of the key which is also used in the unwrapping process.")

    print("\nFor simplicity's sake, we have already defined a value for the key encryption key to be b'1234567890123456'.")

    wrappingKey = b'1234567890123456'# Must be 16,24,32 bytes
    userYN = input("Do you wish to change this value?(y/n):")
    if(userYN == "y"):
        wrappingKey = input("Enter a new value for the wrapping key(MUST BE OF 16, 24 OR 32 BYTES):")
        wrappingKey = wrappingKey.encode('utf-8')
        while (True):
            if((len(wrappingKey) % 8 == 0) and len(wrappingKey) >= 16 and len(wrappingKey) <= 32):
                break
            print("The wrapping key must be a valid AES key length")
            wrappingKey = input("Enter a new value for the wrapping key.(MUST BE OF 16, 24 OR 32 BYTES):")
            wrappingKey = wrappingKey.encode('utf-8')

    print("\nAnd again, for simplicity's sake we've defined the key that is to be wrapped to be b'wow-this-is-fun!'")

    keyToWrap = b'wow-this-is-fun!' # Must be 16,24,32 bytes

    userYN = input("Do you wish to change this value?(y/n):")
    if(userYN == "y"):
        keyToWrap = input("Enter a new value for key to be wrapped.(MUST BE OF 16, 24 OR 32 BYTES):")
        keyToWrap = keyToWrap.encode('utf-8')
        while (True):
            if((len(keyToWrap) % 8 == 0) and len(keyToWrap) >= 16 and len(keyToWrap) <= 32):
                break
            print("The wrapping key must be a valid AES key length")
            keyToWrap = input("Enter a new value for the wrapping key(MUST BE OF 16, 24 OR 32 BYTES):")
            keyToWrap = keyToWrap.encode('utf-8')
```

```
    wrappedKey = keywrap.aes_key_wrap(wrappingKey, keyToWrap,
backend=default_backend())

    print("\nThe encrypted value for the wrapped key is:", wrappedKey)
    input("Press Enter to see how we computed the wrapped key value...")

    print("\nWhat just happened?")
    print("-----------------------------------")
    print("\nSo to go over how we got the encrypted value, lets take a moment to recall all the
components involved.")
    print("First off, when the program asked about setting a key encryption key, this key value is
what's going to remain constant in both wrapping and un-wrapping the new key (Defaulted to
b'1234567890123456' if no input).")
    print("The 'new key' is essentially the symmetric key we are wrapping so that we have a safe
transfer of keys from the sender side to the receiver.")
    print("This 'new key', we will dub it as the 'keyToWrap' value and this is the second value the
program asked for your input. It was defaulted to b'wow-this-is-fun!' if there was no input.")
    print("\nSo now we have the following: ")
    print("1) Key Encryption Key 'wrappinKey' = b'1234567890123456' or whatever you
inputted.")
    print("2) New Symmetric Key 'keyToWrap' = b'wow-this-is-fun!' or whatever you inputted.")
    print("\nFinally, the program used the 'wrappingKey' to encrypt the 'keyToWrap' and that's
how we got our 'WrappedKey' value.")
    print("To clarify, the wrapped key is:", wrappedKey)

    input("Press Enter to see how we can return back to our 'keyToWrap' value...")
    print("\nWhat about un-wrapping?")
    print("-----------------------------------")
    print("\nNow if we wanted to get back our 'keyToWrap' which would be the symmetric key
we wanted to exchange, it is actually quite simple given the parameters and values.")
    print("This is most helpful when trying to decrypt the wrapped key as the receiver of the new
symmetric key.")
    print("\nJust like what we stated above when figuring out our wrapped key value, lets see what
we have:")
    print("1) Key Encrytion key 'wrappingKey' = b'1234567890123456' or whatever it was you
inputted. (REMEMBER THIS PARAMETER REMAINS CONSTANT FOR BOTH
OPERATIONS)")
    print("2) 'wrappedKey' = ", wrappedKey)
```

```
    print("\nJust like before, using the two values and then decrypting using AES, we get back our 'keyToWrap' value")
    c = keywrap.aes_key_unwrap(wrappingKey, wrappedKey, backend=default_backend())

    print("After decrypting we end up with the symmetric key that was to be exchanged between sender and receiver:", c)

#MAIN DRIVER-----------------------------------------------------------

print("=========================================================")
print("Welcome to the Swiss Army Cryptographic Toolset")
print("=========================================================")
print("\nHere are five cryptographic certificates you can learn more about: ")
print("1) 3DES - Choose 1 to perform 3DES")
print("2) AES - Choose 2 to perform AES")
print("3) RSA - Choose 3 to perform RSA")
print("4) HMAC - Choose 4 to perform HMAC")
print("5) AES Key wrapper - Choose 5 to perform AES Key Wrapper")


while(True):
    answer = input("Enter your choice: ")
    if(answer == "1"):
        DES3CIPHER()
        break
    elif(answer == "2"):
        AESCIPHER()
        break
    elif(answer == "3"):
        RSA()
        break
    elif(answer == "4"):
        HMAC1()
        break
    elif(answer == "5"):
        AESKeyWrap()
        break
    else:
        print("Invalid Input! Please try again.")
```

References

- https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/
- http://en.wikipedia.org/wiki/RSA_(cryptosystem)
- https://duckopensource.com/crypto/rsa/
- https://www.dlitz.net/software/pycrypto/api/current/Crypto-module.html