USER MANUAL

Swiss Army cryptographic toolset for beginners

INFR 3600U GROUP 6

BY: AMANDEEP SINGH & TONY WANG

This user manual will cover the five cryptographic tasks implemented within the cryptographic toolset. Down below you can see a screenshot of the main menu screen; from here you can select any of the five tasks that you wish to implement.

```
Welcome to the basic command line - based “Swiss Army cryptographic toolset for beginners”. Different tasks
available are below:
Press 1 to Create a Certificate Signing Request (CSR)
Press 2 to Create a self-signed certificate
Press 3 to Encrypt a symmetric authenticated message via Fernet
Press 4 to Generate a Hash-based message using HMAC
Press 5 to Encrypt a message via AES

Press 0 to exit

Select task number you want to implement:
```

Task 1 – Create a Certificate Signing Request (CSR)

To Create a Certificate, the following libraries listed below are used to make this happen:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes
```

These X.509 certificates are used to authenticate clients and servers; commonly used for web servers that use HTTPS. The Certificate Authority (CA) allows you to obtain a certificate following these steps below:

1. Private/public key pair is generated.
2. Request for a certificate is created; signed by your key to prove its yours.
3. CSR is then given to a CA without the private key.
4. The resource you want a certificate for is validated by the CA.
5. Certificate is then signed and given to you by the CA. (Includes public key & resource)
6. Server is formed to use the certificate; sharing your private key to server traffic.

Getting Started: Firstly, the program generates a secret key and saves it as (key.pem) automatically by using RSA (AES-256) algorithm. Secondly, a CSR request (CSR.pem) that contains some sample user information is generated for further using.

```
Starting the process....
Generating private key....
Writing private key to path ./csr/key.pem file....
Generating CSR....
Writing CSR to path ./csr/csr.pem....
Now you can give CSR to Certificate Authority (CA), who will give certificate to you in return
```

User can also view the public value n and e for RSA algorithm.

```
Would you like to see the public values n and e?
Press 0 for No and 1 for Yes: 1
<RSAPublicNumbers(e=65537,
n=26062147996657551157076977273170878713465951736598259495457133271682317330466143635903394354462041898949087138740 0
010935023363503365262000452410290715562781245341516216513172655729688278753579895962001597209409964125999540089735 78
810180884002475730453545394883482993333897857686103134074028818313440424248454630835484830192076564681366330824043 36
676840843615008983442616392435130593207366739956997721072265254680248894834526145881321042378863141595144883453303 36
984696291088660322351941627259575497430458913519100097528943446561400658628046928144884083718869939936216817590129 07
76651960172752094792084336018614615453 1)>
```

The Key.pem and CSR.pem files can be found in the following path –> ./csr/

csr.pem


```
-----BEGIN CERTIFICATE REQUEST-----
MIIC7zCCAdcCAQAwXDELMAkGA1UEBhMCVVMxCzAJBgNVBAgMAkNBMRYwFAYDVQQH
DA1TYW4gRnJhbmNpc2NvMRMwEQYDVQQKDApNeSBDb21wYW55MRMwEQYDVQQDDApt
eXNpdGUuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtIf8PBgq
dBsHSOIP+iCI0slflFZLnAge4I0XFB8QGcHScumJ79ZK/5vpn4fVxjxT5S57wot4
7bUG4GeWJ3cIslclRFVIiwAx9dnSQ/LVfcHVJLW0i04Ci2ciDvakqa02v3/GyzE0
0zM3enycnE8z+F2dazpabkum6De5rvrxVmAobsqZPhp5EwsKelpxQAkCneBpNktz
79KUfONX8o4LWUBrWJS//p4MBpfDyWcAnz1ZYfg2P93fRdohdlhSPtFKfSGiIZew
ey5duT8y50M6rkRQ7AKn1eHOksPq8ACYXwJa7S0TquYsmyZXaCIi/TyEjIJ13+ng
6qEHRFir6B6V4wIDAQABoE4wTAYJKoZIhvcNAQkOMT8wPTA7BgNVHREENDAyggpt
eXNpdGUuY29tgg53d3cubXlzaXRlLmNvbYIUc3ViZG9tYWluLm15c2l0ZS5jb20w
DQYJKoZIhvcNAQELBQADggEBAJooeBf/xtSrh9Dn4Lmrvqs7Cw8z5nUN/dgxJkeb
3nHOVQRmMjALIvzbx6neNEkd1GQ9+pgwq5UX6HZHb/yFSunzEyyOOHplbF+orGW+
kc2+B4M4CHoocTRvFOSnDJj+1XWbPOKOv43tP330+j1OzoAexFG0joihCFswh4Xe
OYJm2zOI5XQfbL5hTjx5xD2us7cDWDkniXr1bxymwHHFbycbifUruH5XXAPOir7K
/hSswW16w5+AhD8JJQG3US78v/+hCU4xoKfwCKzoeoL10lHhuh1/huXdzmhpkh7z
SqLs9aKBy7Puw8yaoO+klog0raNIyiknYvdeubriL223Z0s=
-----END CERTIFICATE REQUEST-----
```


key.pem


```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-256-CBC,090D11EE6F03D98301CF8D7792A0BFA1

Y833v/vvwTAr+8FdSiv+sPYOwKBVEoklkVfN+18anpCqrvBoFhSCfmk0ogMiPbXK
VzLFwi2d3dvVkiRaH87fUdZwFNct2Q/adzC04IfOrPO9L0Sajzfn9rI54zwAGUrk
F8YNwDxkAzxImq1/nsyL6Os0xM/etan/x/qE3aEJKm3EJjyby36L8eOjaU9J046Z
4pVv34WxCqZFLyFzrZyLoXSua7vP31RNV2LaqNzQwTItFL1bWWdlVfxryvkJ9bDH
MClAs4AJT/7GvZ1hD1bwJe7C2WgNINFX9cylG1tPrQmn8Z8YeXIESoi9Dh8qYnXh
51B6xtdU4HrqzIclSCtyiPFwEM4hc/501fsztko71HDqxR98S7xW/Axs8KQmNQy3
OUPOvdD/zXNFq14+KBq3nGV82/KGsr8JIz/8TBpNNPy8OScvYZfwsdkMGe+Gvcot
7cxNkQxwG702htETu6DDo4LKIJ1IWAU0ph3Sxs6SjOu+pouTRmzvSVEf7/uN3pA3
QZvt6Fs+qPMkDFJ8SlsFN3DQJm6AnUay34tWK/NRvTMQZp1NqQykrowENMnY8xOM
WQuaQfrWAM15W9ATDgiOvUbpJjGrNW6Vrmwldwtr6DuoYIG9R+0mXNC4/XQAcKnA
uEdjhcjQ9mSiWN3LGsG1p8nZUlQciKpNfwhGZSzzvbjWjjVAY4IikuQaiIJnoZTr
TreJLt3XSgEYHyQnfvDDZ7CNxA/16ISbUhRdCtaIsCjCWj/M0B2DKeioWxUwdbq/
iRVHzoOFKDmzlS4v8n/IQ3C++RY5nVeUuU919srdCA1HNdpiKNazCKBp+k1SxdT5
b4aVNeTLSX7pWavb7S5cPVbreed7PxYweEuVxwI2xgUSu1wN4fWVu9BH43GPszoa
VGA19pvKvM7UepFe4SnAbfNEhUdHqqzT7/hNolDAGVSEH2mCK6wnKESUoMzlXHH6
xaa3xA1hZWxp43At8pKUa/4nRHgTNNYffA52hClxv23ygMmQM1Q+gfdsm7DjMrq6
75KDXokacmOEThjvcWeCeI8tCTeJCM6HpDUaTxJz3g4423iX9B+7/AtCDhQh0E7Z
SvCPTLmLbixVQL2THG7kZRfhsH1b0EGT8EijUZLVSIuGSqP4HyHAEWTzkNYAmK4C
5sa4muQ39aPDY6UCqjrx469czC0TKQrqiPA9P2OpTL0CIZLFImgxfOK9KtSEzu+Y
i2OB9EDXtPkEH9289YoHamJXKGn4gM/PK4qifEbCvTIBFShUy8fwyG1ykiAQcAO7
JLFhZ7WuLIMQGnheJv7AZzkrRsrvkx+FsQIWdUpQQd+luq6moeeGbdw5PS48rYdD
tbvsGd+rF5lfJmnruAYSejA+Xoc8UkYj9KeZJEz4H7Krk03/O7HZYst9GTsqeeS8
hlNq1NxbArXB0H8ix2vQm1eY8HrPEkzG3NH8Bx7Aqt0LMqJighjN9a8niytV5+or
cQGr0H6tlzlfy9c3KAeNyoIGd6j9R/q2XoNideaEStR//F7SPI40X/MsDkddLm/2
RdU0LjaogTL0wa5vdB9+/YqYm0+F0gkzyxjhBrc9qjIcdbtCmSb6kVDO7/NYxvnq
-----END RSA PRIVATE KEY-----
```


Task 2 – Create a self-signed certificate
In the previous task, in order to create a CSR, it was required to get it signed by the CA. This is not the case when you create a self-signed certificate. The private key is used to sign it instead by conforming to the assigned public key. However, a self-signed certificate is not trustworthy like a CSR. Although, self-signed certificates are generally used for local testing and can be easily issued effortlessly. Hence, trust is not the main priority for a self-signed certificate.

Getting Started: The function of task 2 is quite similar to task 1, but task 2 generates a CSR certificate (certificate.pem) instead of a CSR request file. Respectively, the private key and certificate can be used for local testing.

The certificate.pem and key.pem files can be found in the following path –> ./self-signed-csr/

```
Starting the process....
Generating private key....
Writing private key to path ./self-signed-csr/key.pem file....
Generating Certificate, and signing with the private key....
Writing Certificate to path ./self-signed-csr/certificate.pem....
Now you have a private key and certificate that can be used for local testing

Would you like to see the public values n and e?
Press 0 for No and 1 for Yes: 1
<RSAPublicNumbers(e=65537,
n=2829721901951926984280480562839246062679909346873560628409085302519462557572170425753679850989860336312637701297274
4405746305858209073534458373238242347010607670144657162198331031088856063466904462876555399160011259073966268435347
2829125246654014856475999243930969673600281333712947560511277085170087753601052046095816327828377066122867616721099
4016938489111024250131411044603283323910051446756669834715642628059528313907550487417905777401873917404383903544403
8590674596585654856687036485946005805625887277435132158198547615599655938204221682659156019779968384604201072597641068
8821010543439047209074175445286951595 3)>
```

Task 3 – Encrypt a symmetric authenticated message via Fernet
Fernet implements symmetric encryption authenticated messages. Thus, the message can't be deployed or read without the secret key. Passwords are also an available option with Fernet if needed. Key derivation functions such as PBKDF2HMAC, Scrypt and bcrypt are capable of making this possible. Down below you can see the main library used for Fernet.

```
from cryptography.fernet import Fernet
```

Getting Started: Task 3 encrypts user messages and display the result by using the secret key, which is preset to "mypassword". Then a decryption process will be implemented and recover the user input.

```
Starting the process....
Using the password as `mypassword` for encryption with fernet....
Generating the key using password....

Enter message that you want to encrypt: mypassword
Encrypting plain text: mypassword....
Encrypted text is: gAAAAABb4rz6Kg--
ssKZAUHb_7e_eB2BpYEzpNYY7FI4dSVkUOoHNGwAtGESpENjQoT8TBRvEuj6P_GcbwT4Id167AYUjVQTvA==
Decrypting the text....
Decrypted text: mypassword
```

Task 4 – Generate a Hash-based message using HMAC

Hash-based message authentication codes allow you to validate the integrity and authenticity of a message. HMACs also let you calculate message authentication codes with a cryptographic hash function paired with a key. Down below you can see the main libraries used for HMAC.

```python
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
```

Getting Started: A secret key is required from user for HMAC hashing. The key can be any size.

```
Enter the key you want to use with HMAC: 23
Generating the HMAC object....
```

The message is then hashed by using the key and the result is displayed.

```
Enter the message that you want to hash: mymessage
The hash of the mymessage is:
b'\x92a\xff\xf5\xbe\xd9X\xdd}J0\x92\xd6\xa0zQ\xbf\xa7*\x8e\x1da_\\\xf1\x9e\\\xf4\xd0\xb9\x9b\x02'
```

Task 5 – Encrypt a message via AES:

In order to encrypt or conceal the sender and receivers' content, the same secret key needs to be used mutually. One disadvantage of Symmetric encryption is that it only provides secrecy but not authenticity. AES (Advanced Encryption Standard) is a great default option for encryption because it is secure and very fast. Down below you can see the main libraries used.

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
```

Getting Started: Task 5 allows users to encrypt and decrypt their input by using AES encryption algorithm. Since AES has a fixed block size of 128 bits, paddings with 0(s) will be added if user's input size that may not be any multiple of 32 bits.

```
Starting the process....
Generating the key and IV for the AES....

Enter the message you want to encrypt: 3423234254546tony test
Since size of plain text is not multiple of block length 32 bytes, adding trailing zeroes to plain text....
Modified plain text: 3423234254546tony test0000000000
```

Then the program demonstrates encrypted and decrypted text, which is the same as the input.

```
Cipher text:
b'\xfd\xd4zt\x86B3\xf5K\xd6;\xe0\xe6q\xe6\xba\xaau\xf8L|o\xeb\x12o\x1c\x12\x90H\xf4p='
Decrypting the cipher text....
Recovered text: 3423234254546tony test
```

For more details, users can type 1 to see the key and IV. Else, screen returns to main menu.

```
Do you want to see the IV and Key of AES algorithm?
Press 0 for No and 1 for Yes: 1
key:  b'@\x1b\x10;g\x8f\x85~xtxB\x17\x95\xbc\x95\xaf{?\xeb00f\xbe\xe5\xa3Ke\x04\xac\xd8#'
iv:  b'\x9a\x90U}Qn0p\xc0F\x05\x0c4\xebJ('
```

Appendix – Source code

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import datetime
import base64
import os
from cryptography.fernet import Fernet

usecase = {
    "create_csr": "Usecase: These X.509 certificates are used to authenticate clients and servers; commonly used for web servers that use HTTPS. The Certificate Authority (CA) allows you to obtain a certificate following these steps below:\n1. Private/public key pair is generated.\n2. Request for a certificate is created; signed by your key to prove its yours.\n3. CSR is then given to a CA without the private key..\n4. The resource you want a certificate for is validated by the CA.\n5. Certificate is then signed and given to you by the CA. (Includes public key & resource)\n6. Server is formed to use the certificate; sharing your private key to server traffic.\n\n",
    "create_self_signed_csr": "Usecase: In the previous task, in order to create a CSR, it was required to get it signed by the CA. This is not the case when you create a self-signed certificate. The private key is used to sign it instead by conforming to the assigned public key. However, a self-signed certificate is not trustworthy like a CSR. Although, self-signed certificates are generally used for local testing and can be easily issued effortlessly. Hence, trust is not the main priority for a self-signed certificate.\n\n",
    "passwords_with_fernet": "Fernet implements symmetric encryption authenticated messages. Thus, the message can’t be deployed or read without the secret key. Passwords are also an available option with Fernet if needed. Key derivation functions such as PBKDF2HMAC, Scrypt and bcrypt are capable of making this possible. Down below you can see the main library used for Fernet.\n\n",
    "hash_based_mac": "Hash-based message authentication codes allow you to validate the integrity and authenticity of a message. HMACs also let you calculate message authentication codes with a cryptographic hash function paired with a key. Down below you can see the main libraries used for HMAC.\n\n",
    "symmetric_aes": "In order to encrypt or conceal the sender and receivers’ content, the same secret key needs to be used mutually. One disadvantage of Symmetric encryption is that it only provides secrecy but not authenticity. AES (Advanced Encryption Standard) is a great default option for encryption because it is secure and very fast. Down below you can see the main libraries used.\n\n",
}
```

```
def create_csr():
   print(usecase["create_csr"])
   print("Starting the process....")
   # Generate our key
   print("Generating private key....")
   key = rsa.generate_private_key(
      public_exponent=65537,
      key_size=2048,
      backend=default_backend()
   )
   # Write our key to disk for safe keeping
   print("Writing private key to path ./csr/key.pem file....")
   with open("./csr/key.pem", "wb") as f:
      f.write(key.private_bytes(
         encoding=serialization.Encoding.PEM,
         format=serialization.PrivateFormat.TraditionalOpenSSL,
         encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),
      ))
   # Generate a CSR
   print("Generating CSR....")
   csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
      # Provide various details about who we are.
      x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
      x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"CA"),
      x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
      x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
      x509.NameAttribute(NameOID.COMMON_NAME, u"mysite.com"),
   ])).add_extension(
      x509.SubjectAlternativeName([
         # Describe what sites we want this certificate for.
         x509.DNSName(u"mysite.com"),
         x509.DNSName(u"www.mysite.com"),
         x509.DNSName(u"subdomain.mysite.com"),
      ]),
      critical=False,
      # Sign the CSR with our private key.
   ).sign(key, hashes.SHA256(), default_backend())
   # Write our CSR out to disk.
   print("Writing CSR to path ./csr/csr.pem....")
   with open("./csr/csr.pem", "wb") as f:
      f.write(csr.public_bytes(serialization.Encoding.PEM))
```

```
    print(
        "Now you can give CSR to Certificate Authority (CA), who will give certificate to you in
return")
    user_input = int(input("Would you like to see the public values n and e?\nPress 0 for No and
1 for Yes: "))
    if user_input:
        print(key.public_key().public_numbers())


def create_self_signed_csr():
    print(usecase["create_self_signed_csr"])
    print("Starting the process....")
    # Generate our key
    print("Generating private key....")
    key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    # Write our key to disk for safe keeping
    print("Writing private key to path ./self-signed-csr/key.pem file....")
    with open("./self-signed-csr/key.pem", "wb") as f:
        f.write(key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.BestAvailableEncryption(b"passphrase"),
        ))
    # Various details about who we are. For a self-signed certificate the
    # subject and issuer are always the same.
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"CA"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"My Company"),
        x509.NameAttribute(NameOID.COMMON_NAME, u"mysite.com"),
    ])
    print("Generating Certificate, and signing with the private key....")
    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
```

```
        key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.datetime.utcnow()
    ).not_valid_after(
        # Our certificate will be valid for 10 days
        datetime.datetime.utcnow() + datetime.timedelta(days=10)
    ).add_extension(
        x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
        critical=False,
        # Sign our certificate with our private key
    ).sign(key, hashes.SHA256(), default_backend())
    # Write our certificate out to disk.
    print("Writing Certificate to path ./self-signed-csr/certificate.pem....")
    with open("./self-signed-csr/certificate.pem", "wb") as f:
        f.write(cert.public_bytes(serialization.Encoding.PEM))
    print("Now you have a private key and certificate that can be used for local testing")
    user_input = int(input("Would you like to see the public values n and e?\nPress 0 for No and
1 for Yes: "))
    if user_input:
        print(key.public_key().public_numbers())


def passwords_with_fernet():
    print(usecase["passwords_with_fernet"])
    print("Starting the process....")
    print("Using the password as `mypassword` for encryption with fernet....")
    password = b"mypassword"
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    print("Generating the key using password....")
    key = base64.urlsafe_b64encode(kdf.derive(password))
    f = Fernet(key)
    plain_text = input("Enter message that you want to encrypt: ")
    print("Encrypting plain text: {0:s}....".format(plain_text))
```

```python
    token = f.encrypt(plain_text.encode('utf-8'))
    print("Encrypted text is: {0:s}\nDecrypting the text....".format(token.decode("utf-8")))
    recovered_text = f.decrypt(token)
    print("Decrypted text: {0:s}".format(recovered_text.decode("utf-8")))


def hash_based_mac():
    print(usecase["hash_based_mac"])
    key = input("Enter the key you want to use with HMAC: ")
    print("Generating the HMAC object....")
    h = hmac.HMAC(key.encode("utf-8"), hashes.SHA256(), backend=default_backend())
    message = input("Enter the message that you want to hash: ")
    h.update(message.encode("utf-8"))
    hash = h.finalize()
    print("The hash of the {0:s} is: ".format(message))
    print(hash)


def symmetric_aes():
    print(usecase["symmetric_aes"])
    print("Starting the process....")
    backend = default_backend()
    print("Generating the key and IV for the AES....")
    key = os.urandom(32)
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
    encryptor = cipher.encryptor()
    plain_text = input("Enter the message you want to encrypt: ")
    if len(plain_text) % 32 != 0:
        print(
            "Since size of plain text is not multiple of block length 32 bytes, adding trailing zeroes to
plain text....")
        plain_text = plain_text + (32 - (len(plain_text) % 32)) * "0"
        print("Modified plain text: {0:s}".format(plain_text))
    cipher_text = encryptor.update(plain_text.encode("utf-8")) + encryptor.finalize()
    print("Cipher text:")
    print(cipher_text)
    print("Decrypting the cipher text....")
    decryptor = cipher.decryptor()
    recovered_text = decryptor.update(cipher_text) + decryptor.finalize()
    recovered_text = recovered_text.decode("utf-8").strip("0")
    print("Recovered text: {0:s}".format(recovered_text))
```

```
    user_input = int(input("Do you want to see the IV and Key of AES algorithm?\nPress 0 for No and 1 for Yes: "))
    if user_input:
        print("key: ", key, "\niv: ", iv)


if __name__ == '__main__':
    tasks = {
        1: create_csr,
        2: create_self_signed_csr,
        3: passwords_with_fernet,
        4: hash_based_mac,
        5: symmetric_aes,
    }
    while True:
        print(
            "\nWelcome to the basic command line - based “Swiss Army cryptographic toolset for beginners”. Different tasks available are below:")
        print(
            "Press 1 to Create a Certificate Signing Request (CSR)\nPress 2 to Create a self-signed certificate\nPress 3 to Encrypt a symmetric authenticated message via Fernet\nPress 4 to Generate a Hash-based message using HMAC\nPress 5 to Encrypt a message via AES\n\nPress 0 to exit")
        number = int(input("Select task number you want to implement: "))
        if number == 0:
            print("Exiting the program....")
            break
        if number not in tasks:
            print("Invalid input try again....")
            continue
        print()
        tasks[number]()
```

Appendix: Web sources

Cryptography.io. (2018). *Tutorial — Cryptography 2.4.dev1 documentation*. [online] Available at: https://cryptography.io/en/latest/x509/tutorial/

Tutorial — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/x509/tutorial/#creating-a-certificate-signing-request-csr

RSA — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/

Tutorial — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/x509/tutorial/#creating-a-self-signed-certificate

Hash-based message authentication codes (HMAC) — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/hazmat/primitives/mac/hmac/

Fernet (symmetric encryption) — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/fernet/

Symmetric encryption — Cryptography 2.4.dev1 documentation. (2018). Retrieved from https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/