NETWORKING PROGRAMMING ASSIGMENT – PYTHON

You are required to build a secure, asynchronous, peer-to-peer chat application. Network communications must via TCP. The flow of the program should be:

1. Start the program
2. Ask the user for the following information:
	1. A port on the machine to act for incoming connections
	2. The name of a file containing a directory of individuals that can be contacted
3. Open Server TCP socket locally on port specified by the user
4. Load user directory (specification below)
	1. This is a JSON serialised dictionary format.
5. Wait for either an incoming network message or a message to be sent, typed by the user
6. If an incoming network message
	1. Connect to the destination using the secure protocol, which must include
		1. The generation of key material should be as shown below
		2. Mutual authentication of the outgoing connection
		3. Establishment of a secure channel using SHA256 HMAC and AES256 CBC Cryptography
	2. Receive Message – Message format below.
	3. Check the control information for consistency
	4. Display received message to the user
	5. Terminate Connection
7. If message to be sent
	1. Wait for the message to be completed – Your choice of end indication (special

character, return key, character sequence)

* 1. Request destination from user. This should be selected from the directory, but you

should also provide a manual option to directly specify IP address, port, and password

* 1. Open socket to destination
	2. Connect to the destination using the secure protocol, which must include
		1. The generation of key material should be as shown below
		2. Mutual authentication outgoing connection
		3. Establishment of a secure channel using SHA256 HMAC and AES256 CBC Cryptography
	3. Send Message – Message format below.
	4. Display transmission success or failure message to user
	5. Terminate Connection
1. Go to step 5.

**Message Formats**

You should use a Python Dictionary message format that can be serialised as follows:

Message = { ‘header’:{

‘msg\_type’: type, ‘crc’: val, ‘timestamp’: UTC\_val },

‘message’: base64\_encoded\_text\_message, ‘security’:{

‘hmac’: {
‘hmac\_type’: val,

‘hmac\_val’: val

}, ‘enc\_type’: val

} }

**User Directory Format**

Type of HMAC in use, SHA256
Type of encryption in use, AES256CBC

You should use a Python List of Dictionaries for the directory format, which can be serialised:

Directory = [
{'username': val, 'password': val, 'port': val, 'ip':' val '}, {'username': val, 'password': val, 'port': val, 'ip':' val '}, ...
]

**Key Material Generation**

Once the common user secret is known and the DHSK is generated, these items should be put through a SHA256 HMAC with the user secret as the password and the DHSK as the input.

DHSK |

 +--------+

 user\_secret --| HMAC |

 | SHA256 |

 +--------+

 |

 +----- enc\_key (32 bytes)

 |

 +--------+

 | SHA256 |

 +--------+

 |

 +----- IV (32 bytes)

 |

 +--------+

 | SHA256 |

 +--------+

 |

 +----- HMAC\_Key (32 bytes)

 |

 +--------+

 | SHA256 |

 +--------+

 |

 +----- CHAP\_SECRET (32 bytes)

The output of the SHA256 hash (including the HMAC) will produce 32 bytes of data (represented as 64 hexadecimal characters). These outputs CAN be used directly with the cryptographic libraries in PyCryptoDome except for the IV, which must only be 16 bytes. Given SHA256 returns 32 bytes, use the first 16 bytes (0-15) of the bytes type object, which can be sliced using list slicing notation or a split function. The SAME IV is used every time a message is encrypted and decrypted

**Additional Requirements and Information**

Before starting to write your application, plan the way you will develop features
Use the user name in the CHAP messages to select the incoming information for the key.

* Passwords should not be used directly for CHAP, HMAC and crypto keys. Utilising a key derivation function will be rewarded with higher marks
* All messages sent and received should be logged with appropriate security to protect their confidentiality and integrity
* All keys MUST be lower case, i.e. header, msg\_type, etc.
* All msg\_type values must be lower case
* Where there is no explicit body (ack or nack) the body value is the serialised version of the NoneType or None
* PDUs MUST be serialised as a string and encoded using utf-8
* You may wish to delay the implementation of the user directory functionality to aid simplicity in building the system. Therefore, you may assume only two parties will use this application if this aids or speeds development. However, a user directory is needed for higher marks.
* As a stretch goal, consider implementing confidentiality and integrity in the user directory file, with appropriate crypto material provided on the command line at programme initialisation
* Any messages received out of a time window of 25 seconds should be ignored
* In the Message header use a 32bit Cyclic Redundancy Check calculation from **the zlib library**
* It is highly recommended that you use an object-oriented approach -- this is not necessary, but higher marks will be awarded for an OO approach.
* You must only use primitive cryptographic libraries and not utilise protocols such as SSL/TLS to provide the security functions.
* It is highly recommended that you use some form of FSM for the management of the communications protocol, this is not necessary, but higher marks will be awarded for an FSM approach.
* Please use comments extensively to explain what your code is doing, and how and why it is doing it.
* Please include a short .txt file that explains any security weakness you think are present in the approach