Python Programming Interview Challenge

Overview The goal of this challenge is to write a small program that handles the parsing and verification of UDP packets.

Assumption:

Many things are unaddressed in the description. So, I have made some assumptions here.

1. The message is the whole packet including everything (signature, CRC, packet num, etc)
2. Max packet size can be 1024 bytes
3. The public key is provided in .bin format which is not standard I am extracting the modulo and exponent from it and then making a public key out of it.
4. For the crc calculation I have made some assumptions too.
   1. The packet is assumed to have structured fields, including Unique Packet ID, Packet Sequence, XOR Key, Number of Checksums, Cyclic Checksums, and Digital Signature.
   2. These fields are used for checksum calculation and verification.

Result:  
Maybe due to my assumptions, all the packets that I received from send are not authorized?

Algorithm:  
This is a single threaded which will accept one request at a time. I have another script where I tried doing it multi-threaded but I ran out of time to test it. I think that will not work for now. I need more time for it,

1. Receives packets over a UDP socket.
2. Validates the structural integrity of each packet.
3. Verifies the digital signature of each packet using RSA cryptography.
   1. Calculate the key from
4. Checks the CRC32 checksum of each packet.
5. Introduces an artificial delay between processing packets.
6. Keeps counters for total packets, unverified packets, and failed checksums.
7. Continuously runs in a loop, processing incoming packets.
8. In detail it is as follows:  
   **Imports and Global Variables:**
   * You import necessary libraries like **socket**, **time**, **getopt**, **json**, **RSA**, **pkcs1\_15**, **sys**, and your custom **Packet** class.
   * You also define global dictionaries **keys** and **binaries**.
9. **Command Line Argument Parsing:**
   * You use **getopt** to parse command line arguments. These arguments are:
     + **-p** or **--port**: The server port.
     + **-d** or **--delay**: The delay (artificial sleep) to introduce between packet processing.
     + **-k** or **--keys**: A JSON string representing keys.
     + **-b** or **--binaries**: A JSON string representing binaries.
10. **Socket Setup:**
    * You set up a UDP socket using **socket.socket()** and bind it to a specific host (127.0.0.1) and port (1337) using **sock.bind()**.
11. **Delay Initialization:**
    * If the **delay** is not specified in the command line arguments (**delay** is initially not present), you set **delay** to a small default value of 0.001.
12. **Packet Processing Loop:**
    * You enter an infinite loop that continuously receives and processes packets.
13. **Packet Reception:**
    * You receive data from the socket with a reasonable buffer size (1024 bytes) using **sock.recvfrom()**.
14. **Packet Initialization:**
    * You create a **Packet** instance from the received data to encapsulate the packet's structure.
15. **Structural Integrity Check:**
    * You verify the packet's structural integrity using the **verify\_structural\_integrity** method of the **Packet** class. If the structure is incorrect, you print an error message.
16. **Signature Verification:**
    * You extract the packet's unique ID, format it, and use it to fetch the corresponding key from the **keys** dictionary.
    * You verify the digital signature of the packet using the key and **Crypto** libraries. If the signature is not valid, you print an error message.
17. **Checksum Verification:**
    * You calculate and verify the CRC32 checksum of the packet using the **verify\_checksum** method of the **Packet** class. If the checksum does not match, you print an error message.
18. **Delay and Counters:**
    * You introduce an artificial delay using **time.sleep** to control the rate of processing.
    * You maintain counters for the total number of processed packets, unverified packets, and failed checksums.
19. **Infinite Loop:**
    * The server runs indefinitely, processing incoming packets in a loop.

Packet: this class encapsulates the handling and verification of data packets with a specific structure and provides useful methods for verification and logging of potential issues.

et class

Top of Form

* **Initialization**: The class takes a binary data input and extracts specific fields from it, including Unique Packet ID, Packet Sequence, XOR Key, Number of Checksums, Cyclic Checksums, and Digital Signature.
* **Display Fields**: It offers a method **display\_fields** to print the extracted fields for inspection.
* **Structural Integrity Verification**: The **verify\_structural\_integrity** method checks if the length of the Cyclic Checksums matches the expected length based on the Number of Checksums. If it doesn't match, it indicates a potential structural issue.
* **Public Key Loading**: It provides methods for loading public keys from files.
* **Signature Verification**: The **verify\_signature** method verifies the digital signature using an RSA public key. If verification fails, it logs details about the failure.
* **Checksum Verification**: The **verify\_checksum** method calculates and verifies a CRC32 checksum. If verification fails, it logs details about the failure.
* **Hash Calculation**: Methods **calculate\_received\_hash** and **calculate\_expected\_hash** calculate SHA-256 hashes for the received packet data and data before signing, respectively.
* **Logging**: Methods **log\_verification\_failure** and **log\_checksum\_failure** log verification and checksum failures to respective log files.
* **CRC32 Calculation**: The **calculate\_crc32** method calculates a CRC32 checksum for specific fields in the packet