Coding Documentation

John Trujillo

Timing Constraint

Due to the 1-second time restriction from the verify script, there were a couple of edits that were made to write to the logs more quickly. Concisely those are: pushing up the time at which the script writes to the log by writing immediately after knowing if a packet is invalid, reducing the number of threads running and reducing the over head by only having signature verification running in a multithreaded environment, while having the checksum verification run in another core given that it's a heavily CPU-bounded process. That way, we achieve parallelism as well concurrency.

It is worth noting however that even in a multicore approach, checksum verification still takes roughly 3 seconds. It is because of this that I am changing the `time.sleep(1)` to `time.sleep(4)`.

IO

Parsing / Reading

In our argument parser, we make it mandatory to provide a packet_id to bin file and packet_id to key file dictionary, while port and delay argument are optional and defaulted to 80 and 0 seconds, respectively. We do this because in order for the server to perform digital signature and checksum verification of each packet, having the files is imperative, while a delay is not. Making port optional has no engineering reason.

Writing

There isn't much to discuss in this section besides this being where we use our *delay* value as well as compose and output the expected strings to their respective files.

Packet Structure

Checking Structure

We don't know two things: (1) if there exists a binary and key file to any parsed packet id and (2) if the user inputted the correct file path. Since we are asked to not block any part of the server, we simply wrap accessing the dictionary and opening the files in a try-block statement. If a packet is not considered valid, it is not appended to the sorted array (Packet Ordering).

Digital Signature Verification

In each key file, a modulus and exponent is given: a 64-byte modulus and a 3-byte exponent. To construct the public key we use `RSA.construct()` and provide it the modulus and the exponent, and use RSA-512, or PKCS1 v1.5, as our encrypting scheme. We hash the entire packet but the digital signature with SHA-256 and compare it to the packet's parsed 64-byte digital signature. If the verification does not pass, then we call on the digital signal writer.

Checksum Verifications

We use the multibyte XOR key to XOR the CRC32 checksum of the packet id's binary file fo reference. To efficiently calculate the checksum we use the zlib implementation of CRC32 to do it, and iteratively do this and compare it to *n* 32-bit word that is parsed from the packet, where *n* is the number of checksums to check for the current packet. If the two values of each iteration of the cyclic checksums are equivalent, then the packet is valid. As mentioned before, the heavy computation of the CRC causes a total execution time from the

Misc.

Concurrency

To implement concurrency we originally thought of using forking, but after a closer look that was definitely a bad idea. The number of forking processes, would be exponential, and further, since a forking process fully copies the stack, we would quickly run out of memory. We thus went with a multithreaded approach, it's a lot lighter on the memory usage and more efficient. To never have a blocking execution, each iteration creates a new thread $\texttt{aw}^n$ — where *n* is the iteration. This newly created thread then creates one additional threads, one runs digital signature verification, while checksum verification runs one a different core. From the first release, we removed the conditional variables, and just have it run sequentially, since that how this CRC scheme was structured to be.

Parallelism

The CRC process was given its own core in order to override CPython's GIL implementation since in a multithreaded process the GIL would never run threads in parallel and thus we needed the CRC process to run in another core to avoid this; and have true parallelism by running on two cores. Both processes, use a dictionary proxy, and a manager because the proxy has no way of knowing when its values are modified. We also use a queue from the multiprocessing library which is thread and process-safe.

Packet Ordering

To have these packets ordered in realtime, it's essential to have an efficient sorting algorithm or data structure. It is because of this, that we went with using a PriorityQueue, where the packet sequence number is the key and the value is the packet itself. There isn't a sorting implementation per se, but when running the get.

Code Design Decisions

Since the packet id from the dump file is not necessarily the same packet id all throughout the dumpfile, it could be the case that we get a mix of packet ids that are interleaved. Therefore, whenever our server detects a different in packet id from the current one, and acknowledges the packet at hand as valid, it will append that packet to that packet id's PriorityQueue. To make this reading and accessing efficient, we implement another dictionary where the key is the packet id and the value is its respective sorted-dictionary.