UDP Packet Server

Red Balloon Challenge. October 9th, 2022.

TORQUE (TAREQ) EL DANDACHI

Go over the design I chose for a server that parses and verifies UDP packets built on top of the Python socket API. The server should be capable of handling concurrent streams with multiple keys and binaries.

1 INTRODUCTION

1.1 Overview

The `Server` class is built on top of the socket API and confiured to recieve connection-less datagrams over IP (User Datagram Protocol). Once a packet is received, information about the packet such as the sequence number, XOR key and signature, is extracted from the packet. The checksum of the packet is computed as packets originating with the same `packet_id` come in. Afterwards the hash of the packet without the signature is compared to the signature to verify. Every `Server` instance is designed to take in 4 different `ServerLoggers`.

2 PACKET HANDLING

2.1 Packet Structure

Each packet is divided into 6 main sections:

(1) `id` (4 bytes): A unique id for each checksummed binary.
(2) `seq_num` (4 bytes): The packet sequence number, monotonically increases until it wraps around at 2^{32}.
(3) `key` (2 bytes): A multibyte repeating XOR Key
(4) `num_chksum` (2 bytes): A multibyte repeating XOR Key
(5) `data` (Variable): XOR'd CRC32 DWORDS
(6) `signature` (64 bytes): RSA 512 SHA-256 Signature

2.2 Packet Arrival

The socket allocates `bufsize` amount of bytes in memory to store arriving datagrams. Then a helper function `validate_packet` separates it out into the structure outlined in section 2.1 and runs the signature and checksum verification. The data is stored in the buffer `packet` in hex format.

2.3 Data Handling Choices

I chose to keep most of the math done on `int` types for consistency and as a result not jump between different types. `packet` is then sliced up into its constituents as an `int`.

The only field that is not decoded as follows is the `key` field which repeats the field twice to get an `int` where all the bits are meaningful. Note that since its a repeating xor key, we can copy the lower 2 bytes to the top 2 bytes and have the same key stored in a 4 byte `int` type.

2.4 Verifying the Checksum

CRC is position dependent, by using `zlib`'s `crc32` method and caching the result into a packet `id` specific portion of memory, indexed by the `seq_num` the checksum corresponds to, we can efficiently store and readout the last checksum values for each *seq_id* when `num_chksum` > 1.

However, we don't want this dictionary to blow up in size and so we can see whether each chksum was used to generate the next chksum and whether it was manually used to be checked which guarantees we will never need it again even when `num_chksum` > 1. This reduces the memory size the dictionary would take up. These two conditions are stored next to the value of the checksum in the form of a tuple in the `chksum` dictionary corresponding to a certain packet `id`.

A better way to do this more efficiently is have a pre-allocated array of a certain width (probably `num_chksum` + 1) and you can remember what the index in the array is the effective first index (so you don't have to move every single element and can take the mod of starting position plus index) and the what that first index in that array should correspond to in terms of *seq_id*. This would be less taxing on memory writes and reads compared to my current implementation. I talk briefly about why I chose to implement a dictionary style instead in section 2.6.

2.5 Verifying the Signature

First, I had to investigate the key stored in `key.bin` as I wasn't sure if the encoding matched my previous exposures to SHA keys. By using `hexdump` on the key, we can see the first 6 bytes are `0x010001`, in other words Fermat's 5th prime $F_4 = 65537_d$, suggesting that is probably is the exponent. This means the rest of the binary file is probably the modulus.

Knowing that, we can extract the `exponent` and `modulus` from the bin file (this happens at server instantiation and is stored in a `Key` object). We can then compare

$$(\texttt{signature})^{(\texttt{exponent})} \mod (\texttt{modulus})$$

to the hash of the entire packet without the signature portion using `hashlib`'s `sha256` method. The `check_hash` verification function is also launched as a separate thread.

2.6 Concurrency & Out of order packets

While the signature verification can be done out of order, verifying the checksums can't. However, either of the two methods discussed in section 2.4 could resolve that issue with some modification. One reason I chose the dictionary style implementation over the fixed size array is it allows us to precompute a huge chunk of new *seq_id* hashes in case packets come in out of order. So if packets $p_i \cdots p_j \cdots p_k$ arrive jumbled up, lets say as $p_i \cdots p_k \cdots p_j$ for some $i < j < k$, then when packet p_k arrives, we can calculate the checksum from the last element we checked $c > i$ to k which includes the checksum for j (we computed the hash for $[c, k]$ and $j \in [c, k]$). As more packets come in, if they are precalculated we can read them out from the dictionary (and once we know we won't need them

anymore, we can remove that entry from memory). That way we don't have to double compute the checksum for some entries in the dictionary.

Depending on how fast XOR-ing vs indexing from memory is, we could also find an optimal ratio of recomputing versus caching (i.e. maybe cache every 4 results because allocating memory and accessing it etc. is 4 times less efficient than just running the checksumming 3 more times). I chose not to do that assessment for this implementation.

Since processes are spawned in threads and given caching, the processes are concurrent. There are multiple optimizations I can think about such as making the checksumming run on multiple threads concurrently with some shared memory buffer.

Handling wrap around can be made by tracking the end of the packet and if the number ever overflows, check whether the next packet with the same `id` and wrapping around `seq_num` makes sense as a continuation packet or as a first packet. As a result, you can distinguish between these two cases.

2.7 Interfacing with `Server`

The `Server` class is instantiated with `hostname` and `port` to bind the socket to. It also needs `keys` and `bins` dictionaries that map packet `id`s to the keyfiles' location and the file binaries' location. An optional `bufsize` parameter can be passed to set the buffer size to store incoming packets in, if not specified, it is set to 1024 by default.

These 5 parameters are all processed arguments to the `server.py` script. You can check them out by running `python server.py -h`. The processing involves fixing the types and adding default values. For `keys` and `bins`, the `Server` class modifies them by remapping the input hex string into an integer for the keys and the values to the file contents.

There are 4 logging interfaces that I put into the `Server` class (mostly because they made my life easier). The two logers that are required for this implementation (`verification_logger` and `checksum_logger`), an `error_logger` and a logger for convenience while building it `debug_logger`. The second two are optional and initialized to not log. Note that if the `error_logger` is not initialized the server will halt operation if an error is raised, otherwise the packet is just thrown out.

3 LOGGING

The `ServerLogger` class is initialized with a name, delay, formatting and default level. The name specifies a way to identify the logger and creates a log file with that name (and a `.log` extension). The delay specifies an amount of time to wait before writing to the log file.

Since we probably want to log multiple types of data into a log file, with each log itself with the same overall format within a log file, having an automatic formatter that runs on an input would streamline that process. I opted for a straight-forward not too fancy option of making the formatting parameter take in an ordered list that correspond to functions that are applied to each element by index. So for instance `formatter = [hex, hex_val, str]` applied on the integers `[24, 25, 26]` will append `"0x18\n19\n26"` to the log file. `hex_val` is a function that returns a hex string dropping the `"0x"` from it.

When the logger is called on an iterable to log, it spawns a thread that runs the formatter on it then logs it into a file after a delay.

The class `NoLogger` is used to represent the absence of a logger, calls to it do nothing.

4 TESTING

I created a separate test file for running my own quick tests (they are not very detailed - a quick workaround really). To test out of order packets, I send it controlled and uncontrolled randomly shuffled packets. Using that I was not only able to debug if the server worked but also how much memory reads/writes happen when things are out of order, and as a result helped me optimize the way I removed things from the dictionaries to prevent recomputation.

Future improvements if I was pursuing the project for a piece of code to be used somewhere include test suite automating packet `id`s and `seq_num`. A way to generate random test data and keys, including data and keys that are mismatched/incorrectly formatted. There are definitely lots of test cases that I would include if this was not a one night project.

5 CHALLENGE FEEDBACK

I thought I would just include any of the things I noticed about the challenge here. It was very fun! I love writing communication protocols, my favorite so far was implementing U2F authentication from scratch over RawHID on a teensy.

I enjoyed how the spec was vague, in my experience that reflects how it often feels like jumping into a project on the packet level - mostly lots of curious digging around until it all snaps in place.

There is a small typo on page 3: "Thank about how" to "Think about how".

I honestly missed the `(2 bytes) # of checksums` labelled in the packet structure the first time I read it. I was wondering if there is a better way to highlight it.

I wasn't quite motivated for the introdution of a delay, I am still not sure if this serves a technical reason. It would be cool if this can be mentioned in this challenge. The motivation this challenge provides makes this coding challenge by far the best company job challenge I have done.

The email mentions 2 challenges, however, there was only one challenge sent. Not sure if this is a bug or an outdated email format.