COMP 112 Networks and Protocols

Student: Hao-Wei (Daniel) Lan

Date: 2019-04-29

Final Project Report – HTTPS Proxy with Caching and TLS Inspection

Overview

For the project, I have implemented a proxy that handles both HTTP and HTTPS connections, with the following features:

- All socket reads and writes are done only when select() determines that the socket is ready for reads or writes, to prevent all kinds of blocking.
    - An additional write buffer maintains a write queue for each socket.
- Non-blocking sockets before the connect() call completes, to better handle bursts of new clients.
    - Sockets are changed back to blocking once connect finishes, thus requiring no change to the rest of the program, which are designed and implemented with blocking sockets in mind.
- TLS inspection and censorship of specified key words in webpages.
    - Replaces existing certificates with those issued by me, making the client think content received is legitimate.
    - Responses encoded with Gzip are decompressed for inspection, and recompressed afterwards (Wikipedia uses gzip).
- Separate store and forward mode for content applicable for TLS inspection (determined via the Content-Type header)
    - Packets not candidates for inspection are forwarded directly, reducing unnecessary latency.
- Caching for both HTTP and HTTPS responses.
    - Simple hash table with uri as key and response as value.

Flexible request and response parsers are designed for the proxy, which are capable of handling partial reads from a socket. Chunked transfer encoding for requests are dechunked to allow for TLS inspection. Currently, chunking functionality is not implemented, so dechunked messages are forwarded with the content length header attached.

Github repository: https://github.com/ASingleCabbage/COMP112Proxy

Instructions to run the proxy should be documented in the readme file of the repository.

Design

The key modules are as follows:

- write_buffer

    - A hash table, destination socket as key and a queue of messages as value. Stores messages pending write before the specified socket is ready to be written to.
- ssl_utils
    - Module containing functions for SSL connections, such as ssl reading and writing, as well as certificate generation.
- response_parser_dynamic & request_parser_dynamic
    - Parsers for requests and responses.
    - Response parser are capable of handling partial messages, and can determine whether a response is complete.
    - Response parser does dechunking on all chunked responses.
- http_header
    - Type used to represent a HTTP header. Struct with a name string and value string.
- double_table
    - Two way indexable hash table, containing connection states. Connection state stores the server and client information (SSL * or socket), connection mode (either HTTP or SSL), and connection state (connecting, reading from server, reading from client).
- cache
    - A hash table with uri as key and responses as values. Capable of handling cases where uri is server absolute path (append with host).
- Inspector
    - Takes in a response and does content modification.
    - Currently capable of handling responses encoded with gzip, but more compression algorithms can be easily supported in the future.

Some external libraries are used for this assignment:

- table.h
    - Hash table implementation by Hanson, form COMP 40.
- seq.h
    - Sequence implementation by Hanson, from COMP 40.
- hash-string.h
    - A fast string hashing algorithm used in the cache to hash the keys.
- pcg_basic .h
    - A fast pseudo-random number generator used to generate certificate serial numbers.
- Libdeflate
    - Compress/decompressing gzipped content
- Picohttpparser
    - Only used the dechunking functionality provided by the library. My own parser also has dechunking functions but weren't thoroughly tested, so the library is used in the last moments to ensure functionality.

Evaluation

Performance testing is done using the latency benchmarking script graciously provided by Annie on Piazza. At the time of testing, the store forward modes haven't been implemented yet, it is expected for future versions to have worse latency and throughput, especially on tests only loading content-type text/http. The results are presented below:

	Small, Cached (HTTP)	Large, Cached (HTTPS)	Throughput, Small	Throughput, Large
No Proxy	0.177	0.861	2kbps	185mbps
Proxy	0.088	0.349	4kbps	458mbps
Proxy % increase	50.405%	59.496%	50.405%	59.496%

While performance isn't a design point for the proxy, it performs surprisingly well in the benchmarks. Both HTTP and HTTPS requests gained significant speedup by having a cache, even with inspection functions imposing a not insignificant overhead. Throughput was increased by having a write buffer, which queues up writes even when the receiver isn't ready to receive.

The benchmarks itself isn't representative of real workloads, as it only simulates a best case scenario (high cache hits, single connection). Further tests aren't carried out due to time constraints, but the proxy is capable of handling normal web browsing loads. POST requests such as forms doesn't work with the proxy, which may indicate that some bugs exist with client to server transmissions, possibly with the request parser.

Reflections

There are multiple instances in this project where I was learning through trial and error, especially on SSL connections and certificate handling. Those times can be very frustrating, as there isn't a clear direction, and documentation for OpenSSL is surprisingly hard to navigate through. However, string parsing is the most challenging component, as its tedious and easy to mess up in C.