Project ISA

Monitoring of SSL connection

Pavel Yadlouski (xyadlo00)

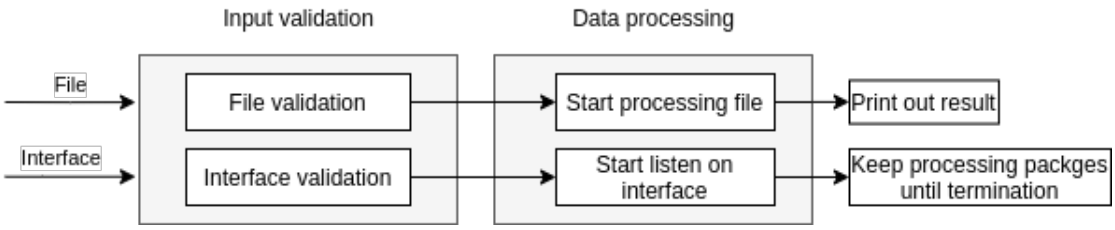
Brno University of Technologies
September, 2020

Contents

1 Introduction

This project aims to create simple CLI application, that can monitor SSL communication. Here monitoring means to aggregate input file **and/or** given network interface.

2 Architecture



The whole program contains two main parts:

1. input validation
   (a) check if file exists and with supported extension
   (b) check if given interface exists and can be opened for listening
2. data processing
   (a) aggregate given file
   (b) start live aggregation of incoming traffic on given interface

The result of file aggregation is written to the standard output as soon as aggregation is completed. If the interface is set, then the program starts listening on the given interface and aggregate incoming packets. When the last packet of any connection is aggregated, then the result of the aggregation for this particular connection would be written to the standard output. In both cases connection is marked as closed and written to standard output if a structure of the connection contains valid SNI attribute and two FIN (from the client and from the server)

2.1 Storing connections

As data structure for storing connections information doubly linked list is used. This data structure was chosen for its simplicity in implementation and use. Also, despite the hash table, the doubly linked list very flexible from the memory usage point of view: the doubly linked list doesn't have a fixed size, so can grow as much as it would be necessary. The case of resizing the hash table requires recalculating hashes for every item, but the doubly linked list doesn't have this problem. There just insert to any side of the list. Also as each element of the list has a pointer to previous and next elements, then deleting of an element is quite trivial: reset pointers between siblings of given connection and delete the required one.

The biggest disadvantage of a doubly linked list is high ($O(n)$) searching complexity.For every package whole list can be looked through. But in this application it is not significant, because is very simple.

2.2 Packet processing

When the program is started and the first packet comes, then it would be set as the first element in the list. After that for every packet would be found a corresponding entry in the list (based on comparing source and destination IP addresses and ports). If the entry doesn't exist, then this packet would be inserted at the beginning of the list with corresponding linking with next element of the list.

Entry for connection is deleted from the list as soon as connection is closed (has valid SNI and two FIN flags). With file aggregation, after process of aggregation is ended, then list is freed.

2.3 Multithreading

Program use separated threads for processing file and interface simultaneously. The main thread would wait first on the file processing thread and then on the interface thread.

3 Implementation

The program contains two parts: checking of input parameters and package aggregation by itself. The first part presents in *sslsniff.c*. Also setup functions for aggregation are called there. The second part is represented in the file *functions.c* with header *functions.h*.

There are several auxiliary data structures for packet aggregation.

3.1 TLS extensions

For processing SSL/TLS extensions there is following structure

```
typedef struct {
    u_int16_t ext_type;
    u_int16_t ext_len;
    char *data;
} extention;
```

This structure helps to go through extensions in SSL/TLS headers and find an extension with SNI. To check the type of an extension there is an element *ext_type*. This value is extracted from the byte stream from corresponding position. Type of extension contains 16 bits, so there is need to convert two following bits, taking into account system byte order. For this purposes (creating unsigned int of 16 bits) there is a function *get_uint_16*

```
u_int16_t get_uint_16(u_int8_t *x){
#if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t tmp[2] = {*(x + 1), *x};
#elif __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t tmp[2] = {*x, *(x + 1)};
#endif
    return *(u_int16_t *)tmp;
}
```

This functions is used whenever we need to convert two following bytes to one value with the right byte order. Attribute *ext_len* is used in *for* loop as dynamic step. It is also created with function *get_uint_16*. Data of extension are stored in *data* attribute.

So, for extracting the SNI name there is a *for* loop with step *ext_len* through all extensions. In this loop, there is *if* statement which checks the type of extension, and if it is equal to 0, then this extension would contain SNI name in its data. In the end, if this type of extension presents, SNI is extracted and stored to the corresponding element in TLS connection structure.

3.2 TLS connection

Each SSL/TLS connection is represented by the following structure

```
typedef struct tls_conn{
    u_int src_ip, dst_ip;
    struct in6_addr ip6_src, ip6_dst;
    u_int16_t src_port, dst_port;
    struct timeval timestamp;
    double duration;
    char *sni;
    u_int packet_count;
    u_int bytes;
    u_int addr_size;
    bool server_fin, client_fin, last_ack;
    struct tls_conn *prev, *next;
} tls_connection;
```

There are the source and destination IP addresses in integer format (for IPv4) and in *ip6_addr* structure (for IPv6). The reason for storing addresses in an integer format is that comparing integers is faster than comparing strings. For comparing IPv6 addresses there is a loop through each byte of address Conversion to human ridable format is made only when the result of aggregation should be written to standard output. Also, for detecting that packet is corresponds to a given connection there are source and destination ports.

For aggregation purposes, there are elements for storing the number of packets in given connection, duration, and bytes. Durations of connections is founded as a result of subtraction *timestamp* (represents timestamp of the first packet aggregated in given connection) and timestamp of a new packet for corresponding connection. For subsection of timestamps, there is function *time_diff*

For interface aggregation end of the communication is detected based on flags *server_fin, client_fin*, and *last_ack*. These flags correspond to TCP flags of a given connection. The connection is considered closed when flag *last_ack* is set to *True*. This happens when all of flags *server_fin* and *client_fin* also have value *True*.

As a doubly linked list is used, then there are also pointers to the next element (connection) and previous.

On the step of input validation, the *cleanup* function is set to be triggered on signal *SIGINT*. This step is necessary for interface aggregation. Processing of live connection is not limited, so this setup insure that data will be freed of interruption.

4 Problems and limitations

4.1 TCP reassembling

Reassembling of TCP packets is not implemented.

4.2 CPU usage

There is a problem that in 1 from cca 10 runs of interface aggregation after a random amount of packets program would load one core up to 100% and stop writing logs to standard output. The reason for this might be some infinite loop in code, but this problem isn't solved.

4.3 Performance

As for storing data during aggregation doubly linked list is used, with a big amount of live connection or on long runs performance of an application is going down due to searching time in the data structure.

References

[1] *SSL Handshake*. Available at: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/.

[2] *TCPDUMP/LIBPCAP public repository*. Available at: https://www.tcpdump.org/.

[3] *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. August 2008. Available at: https://tools.ietf.org/html/rfc5246.

[4] *Transport Layer Security (TLS) Extensions: Extension Definitions*. RFC 6066. Jan 2011. Available at: https://tools.ietf.org/html/rfc6066.