Information Security Workshop - Ex. 4

Eden Koveshi 316221746

March 26, 2019

This exercise intorduces two new components: a *connection table*, and a "*proxy server*" to inspect deeply complicated protocols.

Before using the firewall, run the following command in linux bash,from the working directory:

```
python ./module/interface/proxy.py
```

Part I

Stateful Packet Inspection

Newly added in this exercise is stateful packet filtering, using a *connection table*.

As most good firewalls do, my firewall's connection table is implemented with a *hash table*, using *chaining* method for collision resolution.

The Hash Function

The connection table deploys ***SHA-1*** as it's hash function.

The code for *SHA-1* was taken almost entirely from a free and open source project[1], with mild changes I made to make it fit within the Linux kernel, while being cautious not to make it vulnerable. This implementation has not been validated by the NIST, however it seems good and consistent with the *SHA-1* hashing algorithm.

The reasons I chose *SHA-1* are:

1. **Security.**This is a cryptographic hash function. Those turn out very useful in connection tables. The most common case is *SYN flood*. An attacker can open a connect from (sip,sport) to (dip,dport) only once (second SYN gets thrown away, see the connection table spec.), a good DOS attack would open many connections which will result in the same key (modulus the hash table size). However, hashing the key with a cryptographic, irreversible and collision resistant, hash function prevents this,as

finding a collision in the hash table is equal to finding a collision in the hash function.

As with any good thing, there are caveats:

a. Not too long ago, researchers managed to find a collision in the *SHA-1* function for two different inputs[2], which mathematically means *SHA-1* is broken. However,it is still believed by most to have “strong” security, and used widely in many cryptographic systems.

b. With big effort, SYN flood attack is still possible, using rainbow tables and etc.. Unfortunately I had no time to implement SYN-cookies, but this still solves the most common case, and many different attacks not introduced here.

2. Performence. Cryptographic hash functions usually do a large number of heavy computations, and less suitable for search than regular hash functions (however, a hash table deploying a cryptographic hash function still has better performence than most data structures). *SHA-1* is considered fast among cryptographic hash functions, and performence is important here.

The use of a *hash table* data structure deploying *SHA-1* provides good security and performence together.

The conn interface (*conn.c,conn.h*)

struct conn_t is an abstraction for a *connection*, defined by source ip and port, destination ip and port, state and timeout.

state is a TCP state, and can be one of the next values:

- *TCP_SYN* - this indicates that a SYN packet was sent and the connection is now being initiated.
- *TCP_SYN_ACK* - this indicates that a SYN+ACK was recieved, as a second part of the three-way handshake
- *TCP_ESTABLISHED* - this indicates that the connection is established
- *TCP_ACK* - this indicates that a regular packet (ack bit only) was sent. This applies only to an incoming connection and not to a connection saved in the table.
- *TCP_FIN* - this indicates that a FIN packet was sent, initiating connection termination.

This component defines all connection-related actions that are not dependant of the connection table.

In some cases,when describing a function I'll refer to a conn as a 4-tuple (sip,dip,sport,dport),or 5-tuple (sip,dip,sport,dport,state) where the left-outs can have any arbitrary value in the specific function.

Also, for a connection $c = (sip, dip, sport, dport)$, define it's **reverse connection** by $c^* = (dip, sip, dport, sport)$

struct conn_list_t is a simple list, where every node is of type *conn_t*
The *conn* functions:

int compare_conn(conn_t a,conn_t* b):*

This function compares two coons (a, b, c, d),$(\alpha, \beta, \gamma, \delta)$ coordinate-wise, returns 0 (SUCCESS) iff they are equal

conn_t init_conn(__be32 src_ip,__be16 src_port,__be32 dst_ip,__be16 dst_port):*

This function initializes a new connection by setting it's source port and ip,destination port and ip to the given parameters, and setting it's timeout to [current time in seconds] + 25

conn_list_t init_conn_node(conn_t* conn):*

This function initializes a list node, and sets it's conn member to the given conn param

void destroy_conn_node(conn_list_t toRemove, conn_list_t* prev):*

This function destroys a list node and freeing it's memory while maintaining the list order.

int add_after_conn_node(conn_list_t list,conn_t* new):*

This function adds a new node, containing connection *new*, after list node *list*

conn_t reverse_conn(conn_t* conn):*

Returns the reverse connection of *conn*

int compute_state(conn_t conn,struct tcphdr* tcph):*

For a packet defining connection c, compute c's state by it's flags. The state can be either one of *TCP_SYN, TCP_SYN_ACK or TCP_ACK.* This is an initial state, used for either table insertion or update.

int assign_state(conn_t conn,state_t state):*

Assign state *state* connection *conn.* $((sip, dip, sport, dport, s) \longmapsto (sip, dip, sport, dport, s^*))$

The Connection Table (*conn_table.c,conn_table.h*)

The connection table is an array of size TABLE_SIZE - a constant, currently defined as 50 and may be changed any time, and all of it's elements are of type *conn_list_t** to support chaining inside table cells.

Before describing the functions and API, I'll start with a brief explanation.

Every packet that enters the firewall, except for packets captured by the LOCAL_IN hook, passes a series of stages, regarding the connection table and potentially the rule table.

Packets are categorized by five groups: SYN(no ACK) packets,SYN+ACK packets,RST packets,any other TCP packet, non-TCP packets. Every packet induces a connection, and the next steps are performed on that connection.

•For SYN packets, the stages are: pass rule table -> assign SYN state -> add to connection table

•For SYN+ACK packets, the stages are: look for reverse connection -> make sure it has SYN state -> assign SYN_ACK state -> add to connection table

•For RST packets,the stages are: reverse connection -> remove both connections

•For any other TCP packet: look up for connection and reverse connection in table -> update table accordingly

•For non-TCP packets: compare against the rule table and pass/deny accordingly

If any of the stages fails for a specific packet, it is dropped immediately.

If all stages passed successfully, the packet may continue its journey.

This procedure is defined in the function *inspect_pkt* from last exercise,defined in *fw_rules.c*, which is the main fuction for inspecting packets.

The functions:

conn_t lookup(conn_t* conn,int (*compare_func)(conn_t*,conn_t*)):*

The function looks for the connection *conn* in the table, returns it if found, NULL if not found. It is important since *conn* and the returned connection may have different states. More thorougly, the function uses *compute_idx* (described later) to compute the key of this connection in the table, traverses the list saved in that place, and compares each one of the nodes connection with the input connection *conn*, using *compare_func*.

int remove_conn_from_table(conn_t conn,int (*compare_func)(conn_t*,conn_t*)):*

Given *conn* and *compare_func,* computes the key for *conn* using *compute_idx,* traverses the list at that place and compares every node with *compare_func*, once the node is found, it is deleted using *destroy_conn_node* described above.

int update_table(conn_t cur,conn_t* conn_in_table,conn_t* rev):*

This function accepts 3 parameters, where *cur* and *cur_in_table* both define the same connection, except that *cur* is induced by the current packet, and *cur_in_table* was found in the connection table, their state might differ. *rev_state* is the reverse connection of *cur_in_table*. This function decides whether *cur*'s state is valid or not, and makes appropriate changes, by some rules. Denote their states by s_1, s_2, s_3 wrt. parameters order. The rules are:

• *Threeway handshake:* if s_1=*TCP_ACK*,s_2=*TCP_SYN*,s_3=*TCP_SYN_ACK*, then they form a valid threeway handshake, and s_1 is valid. Update s_2 = s_3 =*TCP_CONN_ESTABLISHED*

• if s_1 =*TCP_ACK* and s_2 = s_3=*TCP_CONN_ESTABLISHED* then s_1 is valid, no update required.

• if s_1= *TCP_ACK* and s_2 = s_3 =*TCP_FIN* then s_1 terminates the connection and it is valid, remove *conn_in_table,rev* from table.

• if s_1=*TCP_FIN* and $s_2 \neq$*TCP_FIN* (or else this side of the connection is already closed and might not send messages), then s_1 is valid. Update s_2 =*TCP_FIN*

As this function applies only to TCP,non-SYN/SYNACK packets, this is all that required to update connections correctly.

Returns 0 (SUCCESS) if any of the rules apply, -1 (ERROR) otherwise

int add_connection(conn_t conn):*

This function computes the key for *conn* using *compute_idx* (surprise..) and inserts it as last in the list lying there (creates a new one if cell is empty)

int compute_idx(conn_t conn):*

Given $(sip, dip, sport, dport)$, computes $s = SHA1(sip||dip||sport||dport)$, where || stands for concatenation.

It then turns s into an integer by splitting it to 5 and summing (mod TABLE_SIZE)

This might harm security, but seems harmless in first sight.

Kernel-to-user-space functions:

*ssize_t show_conn_tab_size(struct device *dev, struct device_attribute *attr, char *buf):*

Returns the size of the connection table, saved in static variable *num_conns*

*ssize_t set_conn(struct device *dev, struct device_attribute *attr, const char *buf, size_t count):*

Sets the static variable *cur_conn_num* to given input

*ssize_t show_conn_tab_size(struct device *dev, struct device_attribute *attr, char *buf):*

Returns a string representing the i'th connection (in linear traverse order)

Those 3 functions are used in user space code to print the entire connection table.

Part II
Deeper Inspection of Application Layer Protocols

Overview

This exercise introduces another new component - deeper inspection of application layer protocols.

This time, we check the data of application layer packets, and as it requires rather complicated computations, this component is implemented as a user-space program, written in python.

The program is a server, with two listening sockets, one on port 8080 - for HTTP packets, and one on port 2021 - for FTP packets.

When a packet arrives, it is inserted into a queue, and checked by http/ftp rules.

To make this work, packets are redirected in kernel code - their source/destination ip and port change so it will be routed to the "proxy" server, and then changed again when coming back from the server. Their checksums are corrected so it won't get dropped in the way.

The "Proxy" Server (*proxy.py*)

The server has two listening sockets, as explained above. The code was inspired by a python socket programming tutorial I've found online[3].

The main function simply creates the two sockets and activates them on two threads, so they can function simultaniously.

class TheServer:

This class implements a server listening on port given as input,and forwards packets to a port given as input.

It has the following functions:

main_loop(self):

This is the main loop of the server,which simply listens and waits to accept incoming connections.

on_accept(self):

This function occurs when accepting an incoming connection. It simply initiates a connection with the "forward" host - the host that should recieve the data eventually (host2)

on_recv(self):

This function occurs when recieving data from the connected socket. This function inspects the packet's data according to HTTP/FTP rules (depends on the recieving server forward port), using *inspect_http/inspect_ftp* functions (described later).

on_close(self):

Defines what happens when the connection closes. Simply closes the connection with the forward host.

inspect_http(self):

This function analyzes the data of an HTTP packet, according to the following rules:

•Headers must begin with HTTP/1 (HTTP/1.0 or HTTP/1.1)

•Content-Length header must exist

•If content length is more than 2000 bytes, ensure it is not an Office file, by looking for the Office magic number in the appropriate offset[4].

If any of these fail, the packet is dropped immediately.

If all tests pass, forward the packet to the forward host.

inspect_ftp(self):

This function analyzes the data of an FTP packet, using a variable *ftp_state* which keeps the current state.

The states are:

- FTP_NONE - FTP connection not initiated yet.
- FTP_USER_SENT - USER command has been sent
- FTP_USER_OK - username has been authorized, waiting for password
- FTP_PASS_SENT - PASS command has been sent.
- FTP_CONN_ESTABLISHED - connection is established.
- FTP_PORT_SENT - PORT command was sent
- FTP_FILE_TRANSFER - In the procedure of file transfer

The function tracks the commands and return codes and makes sure it fits the pattern of $USER- > 331- > PASS- > 230- > (PORT- > 150- > STOR/RETR- > 226) * - > 221$ while allowing $220 OK$ whenever the connection is established, to support any non-get/put command.

Redirections

To make the proxy work correctly, and be "transparent". Redirections are made. Review *redirect_in/redirect_out* to understand the exact redirections. Also, see the example below.Also, checksum is corrected in the end, using the portion of code you gave us (modified a bit[5]).

There are also 3 hooks now - PRE_ROUTING,LOCAL_IN and LOCAL_OUT

PRE_ROUTING and LOCAL_OUT both inspect the packet, and then redirect it. PRE_ROUTING redirects in, LOCAL_OUT redirects out. LOCAL_IN is simply a security/sanity check. All hook functions are implemented in *fw.c*

Redirection example:

A packet (10.0.1.1,p)->(10.0.2.2,80) arrives in PRE_ROUTING, goes through inspection, becomes (10.0.1.1,p)->(10.0.1.3,8080), comes in LOCAL_IN and moved on to proxy, goes through inspection and forwarded in a new packet (10.0.2.3,p)->(10.0.2.2,80), comes in LOCAL_OUT and then becomes (10.0.1.1,p)->(10.0.2.2,80) and then arrives safely at 10.0.2.2

References

1. https://github.com/clibs/sha1
2. https://shattered.io/static/shattered.pdf
3. http://voorloopnul.com/blog/a-python-proxy-in-less-than-100-lines-of-code/
4. https://en.wikipedia.org/wiki/List_of_file_signatures
5. https://stackoverflow.com/questions/16610989/calculating-tcp-checksum-in-a-netfilter-module