Information Security Workshop

Exercise 3

Eden Koveshi 316221746

The project consists of a firewall kernel module and a user interface, where the firewall module is consisted of *list,fw_rules,fw_log* and *fw* objects (source and header files), and the user interface is consisted of *user.c*.

In general, this is a stateless firewall, allowing the user to load custom rules, catching packets and passing/blocking them according to the rules, and logging all traffic going through it.

As stated already, the firewall is a kernel module that consists of two devices - *fw_rules*, a *sysfs device* responsible for all rule-related work, and *fw_log* - resposible for all log-related work. They are patched up together into one kernel module in *fw*.

Note: in the kernel module, *kmalloc's* are used with *GFP_ATOMIC*. This flag is needed in many of them due to context switches during packet catching. I have decided to use it in all of the *kmalloc's* as it works fine for actions not performed during context switches too.

List (list.c,list.h)

This is an implementation of a standard doubly-linked (non-circular) list ADT, where the info of the list nodes is a pointer to a log row.

Note: There are list functions that are not included here since they are irrelevant. However,there are also fucntions that were included but not used as I thought they'd be useful and it turned out they're not. They are submitted as I believe they can help later in the project. They will not be discussed in this document.

struct node_t:

This is a struct representing a list node, containing pointers to next and previous node, and a log row.

node_t* init_node(void):

This function simply initializes an empty node, and returns a pointer to it.

int assign_log(node_t* node,log_row_t* log):

This functions assigns a (non-null) log row to a (non-null) node.

Returns 0 on success and -1 on error.

void destroy_node(node_t* node):
Frees the memory of a given node. If it has a non-empty log row, it frees it's memory as well.

int add_after(node_t* new,node_t* node):
Assigns *new* next in the list right after *node.*
Returns 0 on success and -1 on error.

Rules (fw_rules.c,fw_rules.h)

This part of code is responsible for all rule-related functions, with one exclusion - the hook function is included here as well, as it links rule and log devices, this is also the most major function in this kernel module, and contains all the basic flow.

unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, const struct net_device *out, int (*okfn)(struct sk_buff *)):
As stated before, this is the hook function which links rules and logs, and contains the basic program flow.

This function is registered to a forwarding hook (in fw.c) and does the following:

Upon catching a packet, inspect it's fields and compare them to each one of the rules, to decide whether it should be forwarded along or dropped. A packet may be passed only if a rule matches it, and says to allow it. A packet will be dropped in any one of the following cases: *christsmas packet* - as defined in ex.3 worksheet, firewall is not active, internal error, a matching rule was found ordering to drop it, or no matching rule was found. The former has a priority over the latter, e.g. a christmas packet will be dropped even if it matches an accepting rule.

After a routing decision is made, the packet is logged including the decision and the reason.

Returns NF_ACCEPT or NF_DROP according to the decision made.

This function is protected under spin lock[1] to avoid race conditions.

Note: in contrast to any other rule field, the packet's direction is computed here.

int compare_to_rule(struct sk_buff* skb,rule_t* rule,direction_t dir):
This function extracts the packet's fields out of the network and transport header, using the sk_buff functions.

It compares each one of the extracted fields, against the same field in *rule.*

It also recieves *dir* as it is computed in *hook_func*, and compared here to the rule's direction.

If there are any different fields, aborts and returns -1, otherwise returns 0.

unsigned int is_xmas(struct sk_buff* skb):

This function inspects the packet TCP flags (if there are any) to decide whether or not it's a christmas packet, as defined in ex.3 worksheet.

Returns 1 if it's a christmas packet, 0 otherwise.

decision_t* compare_pkt_against_rules(struct sk_buff* skb,direction_t dir):

This functions compares the given packet against every rule in the rule table.

Aborts when a matching rule is found, or when it completes going over the entire rule table.

Returns a decision packed in *struct decision_t,* which is discussed later on.

decision_t* inspect_pkt(struct sk_buff *skb,direction_t dir):

This function extracts the packet fields and checks if it's a christmas packet and if the firewall is active. If so, sends it to *compare_pkt_against_rules.*

Returns a routing decision.

int append_rule(rule_t* rule):

Adds a new rule to the rule table, pointed by *rule.*

Returns 0 on success and -1 on failure.

void add_localhost(void),void add_prot_other(void):

Add built-in rules allowing localhost, and communication based on protocols not in *prot_t* struct.

void clear_rules(void):

Clears the rule table and frees it's memory.

int string_to_ip(char** ip_string,unsigned int* ip,unsigned int* mask,unsigned int* mask_size):

This function recieves an ip in conventional dotted human-readable format (a.b.c.d/x) pointed by *ip_string* and converts it to integers. Integer ip is stored in *ip,*mask in *mask,* and size of the mask in *mask_size.*

This is done with standard string manipulation, using *strsep* to separate ip from mask and then each one of the bytes.

Returns 0 on success and -1 on failure.

int parse_protocol(char* prot,rule_t* rule):

This function parses a string repesenting a protocol, pointed by *prot* and assigns *rule*'s protocol accordingly.

Returns 0 on success, -1 otherwise.

int parse_port(char* port, rule_t* rule,int flag):

Same as *parse_protocol,* this time with a port. Argument *flag* indicates whether it is *rule*'s source or destination port.

Returns 0 on success, -1 otherwise.

int parse_rule_string(char* rule_string,rule_t* rule):

This function recieves a string representing a rule in the format explained in ex.3 worksheet, and parses it into a rule.

Makes use of the 3 parsing functions described above.

Returns 0 on success, -1 on failure.

log_row_t* create_log(struct sk_buff* skb,decision_t* res,unsigned char hooknum):

This function recieves an sk_buff representing a packet header, pointed by *skb*, a routing decision *res* that was made for the packet, and the hooknum that caught the packet, and creates a suitable log row for it.

Returns a pointer to the log row, or NULL on error.

This also includes *timestamping*[2] the log.

Attribute functions, allowing communication between user and kernel space:

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

This is the *show* function for the *fw_rules' rules_size* attribute discussed later.

Returns the number of rules in the rule table.

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

This is the *show* function for the *fw_rules' active* attribute discussed later.

Returns 1 if the firewall is active, 0 otherwise.

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

This is the *store* function for the *fw_rules' active* attribute discussed later.

Sets the firewall state to active if it's given 1 as input, or inactive if it is given 0 as input. In any other case, firewall state will not be changed.

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

This is the *store* function for the *fw_rules' add_rule* attribute discussed later.

Recieves a string, representing a rule by the format explained in ex.3 worksheet, and puts it in the table.

If the string is not matching the rule format, it is ignored.

ssize_t clear_rules_store(struct device *dev,struct device_attribute *attr,const char* buf,size_t count):
This is the *store* function for the *fw_rules' clear_rules* attribute discussed later.
Clears the rule table completely and frees memory of all rules, except for the two default rules (localhost and prot_other).

ssize_t set_cur_rule(struct device *dev,struct device_attribute *attr,const char* buf,size_t count):
This is the *store* function for the *fw_rules' show_rules* attribute discussed later.
Sets the number of rule to read on the next reading from the attribute file.

ssize_t get_rule(struct device *dev, struct device_attribute *attr, char *buf):
This is the *show* function for the *fw_rules' show_rules* attribute discussed later.
This functions takes the *n-th* rule, where *n* is the number recieved by last call to *set_cur_rule* (0 on default) and parses it into rule format.

Log (fw_log.c, fw_log.h)

This part of the module is responsible for log-related functionality. It maintains a list of logs, using the functions described in *list* section, and saves a pointer to the head and tail nodes.
node_t* add_log(log_row_t* log):
Adds a given log to the list.
Returns a pointer the node in which it was inserted, or NULL on error.

int compare_logs(log_row_t* a,log_row_t* b):
This function checks whether two logs are equal or not, by comparing all of their fields, except for timestamp and log count.
Returns 0 on match, -1 otherwise.

node_t* find_log(log_row_t* log):
Searches for the given log in the list of logs.
If found, returns a pointer to the node in which it resides.
Returns NULL in any other case.

node_t* find_node_by_log(node_t* start,log_row_t* log,int (*compare_func)(log_row_t*,log_row_t*)):
This function searches for a given log in the list of logs, starting from *start* and comparing two logs using a comparison function *compare_func.* The comparison function is expected to return 0 iff the logs are equal.

If found, returns a pointer to the node in which it resides.
Returns NULL in any other case.

int remove_log(log_row_t* log):
Removes a log from the list.
Returns 0 on success, -1 on failure.

int log_pkt(log_row_t* log):
This function does the main work in this part of the module. It recieves a log describing a packet and:
1) Searches for it in the log list.
2) If found:
  2.1) Incerements log count and updates timestamp.
3) If not found:
  3.1) Adds it to the list, and updates the list tail.
This function is protected under spin lock too.
Returns 0 on success and -1 on failure.

void clear_logs(void):
Clears the log list and frees it's memory.

int snprintf(char *buf, size_t size, const char *fmt, ...)[3]***:***
An implementation of user-space library function *snprintf.*
Returns number of formatted elements.

int ip_to_string(unsigned int ip,unsigned char* s)[4]***:***
Convers an ip given in integer form to a human readable format.
Returns 0 on failure and a positive number on success.

Attribute related functions:

ssize_t num_logs_show(struct device *dev, struct device_attribute *attr, char *buf):
This is the *show* function for the *fw_log's log_size* attribute.
Returns number of logs in the log list.

ssize_t clear_logs_store(struct device *dev, struct device_attribute *attr, const char *buf, size_t count):
This is the *store* function for the *fw_log's clear_logs* attribute.
It recieves any *byte,* and clears all logs from memory.

Device related functions:

int show_logs(struct file* file,char __user* buffer,size_t length,loff_t* offset):
This is the *fw_log* device *read* function.

It returns a detailed description of all logs in the list, or the constant string *"ERROR"* on error.

int device_ open(struct inode* inode,struct file* file),int device_ release(struct inode* inode,struct file* file):
Do nothing, simply to override the default function.

FW (fw.c,fw.h)

This function is resposible for module and device creation.
struct decision_ t:
This struct has two fields - action and reason. Action can be either NF_ACCEPT or NF_DROP and reason can be any valid reason to accept/block a packet.

static char *log_ devnode(struct device *dev, umode_ t *mode)[5]:
This function defines the permissions of the log device. Currently set to 0666.

module init function:
The init function registers the log as char device, given the functions explained in it's section.It then creates a sysfs class and two devices under it, one for the log and one for the rules.
It then registers all of the attributes, and the hook.

module exit fucntion:
unregisters all registered items, and frees all memory, including rules and logs.

About the attributes:
•*fw_ log/log_ size:* Returns number of logs maintained in the log list. Has reading permissions only and linked to *num_ logs_ show* function.
•*fw_ log/log_ clear:* Has writing permissions only and linked to *clear_ logs_ store* function. Invokes *clear_ logs* upon writing.
•*fw_ rules/active:* Has reading and writing permissions, and linked to *active_ show* and *active_ store* functions.
•*fw_ rules/rules_ size:* Has reading permissions only, and linked to *num_ rules_ show* functions. Returns number of rules in the rule table.
•*fw_ rules/add_ rule:* Has writing permissions only, and linked to *rule_ store* function. Recieves a string in the rule format defined, and adds a suitable rule to the table.
•*fw_ rules/clear_ rules:* Has writing permissions only, and linked to *clear_ rules_ store* function. Upon writing, clears all rules.

•*fw_rules/show_rules:* Has reading and writing permissions. Writing updates the rules device static variable *cur_rule_num*, and reading returns the rule at location *cur_rule_num*.

User-space program (user.c)

There is really no much to say about the user space program, it parses the user input and uses the attributes to get the work done.

About the Makefiles[6]

user-program makefile is a standard gcc call, compiling it into main.

The kernel module makefile, defines firewall.ko as an object and dependancies, resulting in all code compliling into one module.

References

[1] - https://www.kernel.org/doc/Documentation/locking/spinlocks.txt
[2] - https://stackoverflow.com/questions/8653839/human-readable-timestamp-in-linux-kernel
[3] - https://stackoverflow.com/questions/12264291/is-there-a-c-function-like-sprintf-in-the-linux-kernel
[4] - https://stackoverflow.com/questions/1680365/integer-to-ip-address-c
[5] - https://stackoverflow.com/questions/11846594/how-can-i-programmatically-set-permissions-on-my-char-device
[6] - https://stackoverflow.com/questions/349811/how-to-arrange-a-makefile-to-compile-a-kernel-module-with-multiple-c-files