# Creating A Minix Firewall

Thomas Peterson
thpeter@kth.se

December 16, 2019

# 1 Introduction

A common way for attackers to get access to remote computers is by abusing insecure network connections. In this project, a simple firewall was created for the MINIX 3 operating system to counter the basic ways of abusing network connections. MINIX 3 is a microkernel based operating system whose kernel consists of under 5000 lines of code[1], resulting in it being one of the smallest available kernels[2]. MINIX 3 is an open-source operating system designed to be highly reliable, flexible, and secure which is available for free[3]. The main use cases of MINIX 3 is for embedded systems and for education[4]. MINIX 3 achieves its reliability, flexibility and security through limiting the kernel land to only contain the microkernel and placing the rest of the operating system in user land[3].

The main goal of the firewall was to intercept all traffic of the host and filter it according to a set of rules as well as inspecting the traffic for port scans and DOS attacks. To enable flexibility, and for enabling other entities in MINIX 3 to ask for filtering, the firewall was implemented as a separate server. Since the server's main functionality was to *decide* if packets should be dropped or kept, it was referred to as the "firewall decision server" and was consequently named fwdec in MINIX 3.

MINIX 3 uses the lightweight IP stack(LWIP) developed by Adam Dunkels implemented as a server as well[5]. The goal of LWIP is to be a minimal TCP/IP stack using as little memory as possible and having a minimal code size, thus being a good fit for MINIX 3 as MINIX 3 is purposedly kept as small as possible to achieve its goals. All packet traffic from and to a MINIX 3 host must pass through the LWIP stack and consequently this part of the OS could be considered a good place for packet interception.

# 2 Vulnerabilities Countered

The firewall can drop traffic from and to different ports and IP-addresses based on a set of rules. Some examples of vulnerabilities that can be countered with this basic filtering is unauthorized access to specific ports on our host and malware that tries to send packets from our host. Additionally, the firewall has a minimal stateful functionality for detecting and blocking TCP SYN scans as well as TCP SYN flood attacks. It is thus resistant to port scanning and DOS attacks based on these two techniques.

Depending on the configurations of the firewall, it may or may not be able to protect against port scans and DOS attacks. For example, it is able to protect against scans and DOS attacks if all traffic comes from an unathorized IP as all this traffic will be dropped. Additionally, it is able to protect against SYN-based scans and DOS attacks. However, if the host uses other DOS or scan techniques than this, the firewall could not automatically detect and prevent the attacks. Instead, a human operator would have to detect the attack and manually alter the filtering rules of the firewall to drop the packets of the attacker.

# 3 Related Works

During the Google Summer of Code 2010, Stefano Cordio attempted to port the NETBSD firewall to MINIX 3[6]. This project was planned to be conducted in two steps. The first would be to port the NETBSD packet filter interface (pfil) and the second to port the packet filter (pf) module. The idea was thereafter to create interception hooks with pfil in LWIP and specify callback functions for these hooks using pf. It should however be noted that this project has not been updated past week 9[6]. Additionally, the wiki states that the project needs a massive update and from the MINIX 3 github repository[1] it can be determined that the firewall has not yet been merged into the latest MINIX 3 distribution. It thus appears to be likely that this project has been abandoned.

Another attempt at porting a firewall in MINIX 3 was MinixWall, a project started in 2007. Similarly to Cordio's project, this project also attempts to port an already existing firewall. MinixWall is a port of the iptables/netfilter firewall framework of the linux kernel freed from linux-specific datastructures[1]. This project was deemed successful[1] but does not either appear to have been merged into the official MINIX 3 distribution. Rather it seems like MinixWall was continuously developed until 2012 in its own separate github repository[2].

Both of these projects differ from the firewall created in this work in the sense that our firewall is built from scratch rather than ported from another operating system. This could have the advantage of facilitating the development as one could avoid having to select what part of another software is to be included in MINIX 3 and subsequently run into dependency issues.

# 4 Design Choice

There exists a variety of ways to implement a firewall. In this section, the design of the firewall is explained and discussed. First, packet interception is described. Thereafter, a brief description of the firewall decision server is provided. Finally, a section is devoted to the communication between the servers with additional technicalities.

## 4.1 Packet Interception

As mentioned earlier, MINIX 3 uses LWIP as its TCP/IP stack. LWIP consists of 4 layers[5]. At the lowest abstraction level is the network interface level, consisting of interfaces to different networks. Thereafter comes the internetwork layer which treats IP packets. Then, the packets reach the transport layer which treats transport level protocols such as UDP and TCP. Finally, in the packet content reaches the application at the application level. In the internetwork layer there is one choke point for all incoming ipv4 traffic, the function

[1] `https://github.com/Stichting-MINIX-Research-Foundation/minix`
[2] `https://github.com/booster23/minixwall`

*ip_input()*. Additionally, there is a corresponding choke point for all outgoing ipv4 traffic, the function *ip_output_if()*. Both of these functions are defined in the file *ip4.c* located in */minix/lib/liblwip/dist/src/core/ipv4/*. The packet interception is conducted at this level due to the choke points enabling interception in only two spots rather than considerably more spots.

In fact, if the interception wouldn't have been conducted at the internetwork level of the LWIP stack but rather in the transport layer, the packets would have had to be intercepted in different parts of the code for each protocol. As there are a total of 256 possible transport layer protocols and each protocol may require interception for outgoing and incoming traffic, this could mean up to 502 interception points[7]. However, a downside of intercepting at the internetwork level is that one can not have access to the already parsed packet fields as they are parsed in the transport layer. Thus, it is required to manually extract the relevant fields for the filtering process from the packet data. Another thing to note is that only ip version 4 is captured through the two functions mentioned above and consequently that ip version 6 is excluded from the interception. Before releasing the firewall to the public, interception of ip6 packets should be added as well. Otherwise, it could be possible for attackers to bypass the filter/firewall by simply using ip6 packets rather than ip4.

### 4.2 The Firewall Decision Server

The firewall decision server(fwdec) is implemented as a MINIX 3 server and consequently has access to the inter process communication (IPC) service provided by the kernel. The fwdec server waits until it receives an IPC message containing information about a packet from another server. This information constitutes of the packet's protocol type, source ip, source port, destination ip, destination port and transport layer flags.

Once it has received an IPC message with packet information, the firewall looks for bad traffic patterns indicating malicious behaviour and then applies a set of filtering rules to the packet information. If the packet matches one of the rules and the firewall is in black list mode, the firewall sends a message back to the sender indicating that the packet should be dropped. Similarly, if the packet does not match a rule but the firewall is in whitelist mode, a message to drop the packet will be sent to the sender. In all other cases, the sender will be told to keep the packet. Apart from dropping the packet, the firewall decision server also logs the drop as the packet information together with the reason for the drop. This information can be useful for system administrators to diagnose if the system is being attacked or simply to check that the filtering rules have been configured correctly.

### 4.3 Communication

Since both the LWIP stack and the decision server are implemented as servers, they can communicate using the IPC functionality provided by the kernel[8]. There are three types of IPC operations in minix[8]. The first is the SEND operation which sends a message and blocks the sender until the message has been sent. The second is the RECEIVE command

which blocks the invoking entity until a message is received. Finally, the last IPC operation is SENDREC which sends a message and blocks the sender until a reply is received. In this work, the SENDREC operation was used to send the packet information to the firewall and force LWIP to wait for a reply.

Important to note is that IPC communication handles messages with a payload size of 56 bytes[8]. If more memory must be sent, one can use memory grants. However, as 56 byte was enough for the parameters the firewall needed, memory grants were not required for the communication. However, if one would like to have a more complex stateful firewall, it could be useful to send a memory grant for the whole packet so that the firewall could access the whole packet's content and thus conduct deeper analysis.

The whole process of filtering can be observed in figure 1. The interception of packets starts at the two leftmost functions of the figure. These functions are invoked depending on if the packet belongs to incoming or outgoing traffic. Once a packet has been captured, the *pbuf_filter* function is called. The *pbuf_filter* function extracts the relevant information from the packet header and subsequently calls *fwdec_check_packet*. The *fwdec_check_packet* function then prepares the IPC message and calls *do_invoke_fwdec* which then sends the message to the firewall using SENDREC.

The firewall's *get_work* function receives the message, checks the message type and calls the corresponding function, which in this case is *check_packet*. The *check_packet* function filters the packet and concludes if it should be kept or dropped. Thereafter, it calls *reply* which then send the response to LWIP using the SEND operation. The decision then reaches the left most two functions which then take actions based on the firewall's response.

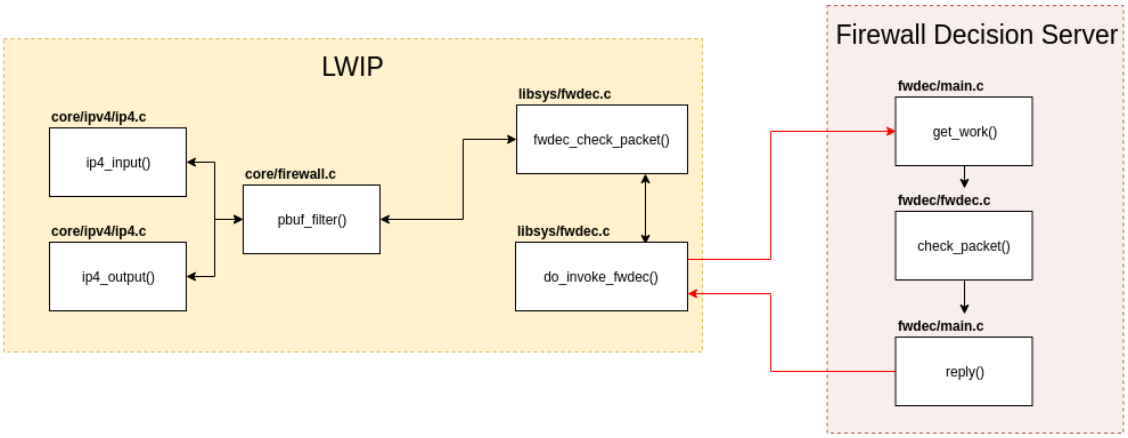


Figure 1: Communication between LWIP and the Firewall Decision Server. The folders *core*, *libsys*, and *fwdec* corresponds to */minix/lib/liblwip/dist/src/core*, */minix/lib/libsys* and */minix/servers/fwdec respectively*. Black arrows denote function calls or returns from function calls while red arrows denote IPC messages

One could argue that *pbuf_filter*, *fwdec_check_packet* and *do_invoke_fwdec* could have been written as one function. The reason for not doing this was to make the design more appropriate for future implementations. For example, as *do_invoke_fwdec* takes an arbitrary message and sends it to the decision server, future implementations could reuse this function to communicate with the decision server. Similarly, if the relevant packet information has already been obtained, one would not need to execute the code in *pbuf_filter* but could simply call *fwdec_check_packet* directly.

# 5 Contributions

The majority of my contributions were concerned with the logic of the firewall decision server(logging, filtering, IPC message handling, DOS protection among other things), finding good packet interception places in LWIP, determining how to drop packets as well as establishing communication between LWIP and the firewall decision server. For the IPC communication and LWIP packet interception, I worked closely with Davis. Finally, concerning the statefulness functionality of the firewall, me and Emelie focused on the implementation of the SYN scan and SYN flooding protection while Marcus and Davis designed and setup the tests for this functionality. This section presents more details concerning the tasks I contributed to.

## 5.1 Rule-Based Filtering

As mentioned earlier, the firewall filters based on rules. Each time it receives a packet, it will try to match the packet to its set of rules. The firewall can be put in two different modes which handles rules differently. These are the black list and white list mode. In white list mode, a packet is kept only if it matches at least one rule. Conversely, in black list mode, a packet is kept only if it does not match any rule.

Each rule consist of protocol, source ip, source port, destination ip and destination port fields. If a field has a non-zero value, the packet's corresponding field matches this field only if the packet's field has the exact same value. If the rule has a field equal to 0 it means that it will match any value. If all fields of the packet matches all fields of a rule, the packet is said to match that rule. It should be noted that the ip-addresses do not yet use subnet masks and consequently that it is currently impractical to try to apply a rule to a large subnet.

Currently, the rules are stored in the heap as a linked list where each rules has a pointer to the next rule. For each rule, the firewall allocates the required space on the heap. The reason for using dynamic memory rather than static is that there won't be any limit on the number of rules that can be used and thus it would be easier to add support for adding rules during run-time in the future.

Finally, it is important to note that there exists a protocol named UDP-lite which is also handled by LWIP[5]. This protocol differs from the UDP protocol in the way that it allows partial checksums rather than the full or no checksum alternatives typical for the UDP protocol[9]. Since these two protocols have different protocol numbers, rules with one protocol's protocol number won't match packets containing the other protocol's protocol number. It was chosen to not combine these two protocol numbers as one to allow for greater flexibility. However, system administrators should bear in mind that UDP rules won't match UDP-lite rules(and vice versa) when setting rules. Otherwise, it could, for example, be possible to access unauthorized ports by using UDP-lite instead of UDP.

### 5.2 Logging

An additional feature that was added to fwdec is the logging functionality. The firewall logs all dropped packets into */var/log/fwdec*[3]. Each log entry contains the reason its corresponding packet was dropped together with the protocol, source ip, source port, destination ip and destination port of the packet. If the packet was blocked by the DOS/Scan protection, the reason will be marked as "suspicious TCP behaviour". Otherwise, when the firewall is in black list mode, the reason will be a rule match and when the firewall is in blacklist mode, the reason will be that the packet didn't match any rule.

An additional minor feature I added towards the end of the project was logging of the mode of the firewall together with its set of rules. Thus, if the firewall is in black list mode one can quickly see which rule the packet violated. Similarly, if the firewall is in white list mode it is easy to compare the rules with the packet information to determine why the packet wasn't kept. All this information makes it easier for a human operator to diagnoze misconfigurations or detect potential attacks. To further enabling debugging, the constant FWDEC_DEBUG[4] can be set to 1. Doing so will print information about each intercepted packet to stdout.

### 5.3 IPC Message Design

As mentioned earlier, IPC messages were used rather than memory grants. To be able to use messages, a new message layout named m_fw_filter was added in ipc.h[5]. This message layout was used for sending filter requests to the firewall decision server and contained fields for the packet information required for the filtering as well as a field denoted as message type. The type field can currently only be set to FWDEC_CHECK_PACKET, signifying a request for the firewall to filter the packet based on the packet information in the message. The type field was included to be consistent with other message layouts but also to facilitate the process of adding additional message types in the future.

[3]The location of the log file can be changed by modifying the constant LOGFILE in */minix/servers/fwdec/fwdec.c*

[4]Located in */minix/servers/fwdec/fwdec.h*

[5]Located in */minix/include/minix/*

### 5.4 Detection of SYN Scans and SYN Floods

A SYN scan is based on TCP and works by sending TCP SYN packet to ports of a target host. If the host responds with a TCP SYN-ACK packet for a specific port, the attacker knows that this port is open. Otherwise, if the host responds with a RST packet, the attacker knows that the port is closed. By sending a large number of SYN packets to different ports and inspecting the responses, an attacker can enumerate the listening ports of the victim machine.

A SYN flood is a type of DOS attack which is, similarly to the SYN scan, also centered around the use of SYN packets. The idea behind a SYN flood is to create a large amount of new TCP connections to a specific port on a victim host. If the victim is unlucky, all the faked connections will fill up the local TCP connection table of the server and consequently, legitimate connection requests won't be taken into consideration by the server.

To counter SYN scans and SYN flood attacks, the firewall server keeps track of TCP SYN and ACK packets in a linked list. Each entry in this list has a field for the corresponding source ip of the packet, a timestamp indicating when the last packet from this ip was received and a SYN counter. Every time the host receives a SYN packet, it increases the SYN counter by one. Conversely, every time it receives an ACK packet, the SYN counter is decreased by one. This is motivated by the structure of the TCP handshake. If an open port receives a SYN packet but never receives a subsequent ACK packet, this should be a warning sign and the counter should be increased. Similarly, if a TCP connection is attempted to a closed port, no subsequent ACK should be received and the counter should be increased. For legitimate TCP connections, the client will always complete the handshake by sending an ACK after the SYN packet. Thus, suspicious connection requests will increase the SYN counter while legitimate connections wiill leave the counter unaffected.

Before altering the entry in the linked list, the fwdec server checks if the count is over a fixed value[6]. If the count has already surpassed this value, the packet is dropped. An additional feature implemented is a time-based reset of the SYN counts of different entries. This is good to have as users might accidentally make too many bad connections over a long period of time, in which case, they shouldn't be blacklisted. When a TCP packet has been received, and its corresponding entry in the list has been found, the server checks the timestamp value. If the difference between the timestamp and the current time is larger than some specified constant[7], the SYN count is reset to 0.

However, if the attacker knows about this mechanism, he might be able to do a SYN scan together with ACK packets after every SYN packet. The ACK packets would trick the firewall into thinking that the connections were established and thus avoiding the TCP protection mechanism. This could be solved by using IPC messages to actually confirm that services were running on the targeted ports. Another issue is that an attacker might

[6]This value can be set by modifying *TCP_MAX_SYNCOUNT* in */minix/servers/fwdec/fwdec.c*

[7]More specifically, the constant *TCP_PROTECTION_TIMEOUT* specified at the start of */minix/servers/fwdec/fwdec.c*

be able to spoof the source ip of the packets and thus trick the firewall into blacklisting the ip of a legitimate users. Even if this attack would require the attacker to know the specific IP of the targeted user, this limitation of the firewall should still be considered as something of importance to fix in future work. A final issue is that the linked list can not decrease in size and thus, an attacker could spoof source ips to make the list very large and consequently slow down the server. This could easily be fixed by freeing entries whose SYN counts reaches 0. However, it was left as future work because of time constraints.

## References


[1] R. Weis, B. Schüler, and S. A. Flemming, "Towards secure and reliable firewall systems based on minix3." in *Sicherheit*, 2010, pp. 85–92.

[2] J. N. Herder, H. Bos, and A. S. Tanenbaum, "A lightweight method for building reliable operating systems despite unreliable device drivers," *Technical Report IR-CS-018, Vrije Universiteit, Amsterdam*, 2006.

[3] MINIX 3 Wiki. (2018) What Is MINIX 3? [Online]. Available: http://www.minix3.org/ (Accessed 2018-12-28).

[4] Corbet. (2005) Minix 3 hits the net. [Online]. Available: https://lwn.net/Articles/156828/ (Accessed 2018-12-28).

[5] A. Dunkels, "Design and implementation of the lwip tcp/ip stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[6] S. Cordio, "Minix 3 firewall project," https://wiki.minix3.org/doku.php?id=soc:2010:firewall, 2010, google Summer of Code 2010.

[7] IANA. (2017) Protocol Numbers. [Online]. Available: http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml (Accessed 2018-12-29).

[8] MINIX 3 Wiki. (2016) Message Passing. [Online]. Available: https://wiki.minix3.org/doku.php?id=developersguide:messagepassing (Accessed 2018-12-29).

[9] IETF. (2004) The Lightweight User Datagram Protocol (UDP-Lite). [Online]. Available: https://tools.ietf.org/html/rfc3828 (Accessed 2019-01-01).