Per Process Firewall in Minix OS

Erik Johansson
erikjo9@kth.se
Group 10

January 2020

1 Introduction

In an age of the internet, billions of devices around the globe are connected and accessible remotely over a network connection (https://www.ericsson.com/en/mobility-report/internet-of-things-forecast). This means that attackers are given the possibility to perform exploits and gain or even prevent access to these devices. In order to mitigate an attackers possibility of sending malicious network traffic the concept of a firewall was introduced. A firewall is a low level mechanism that allows or disallows certain network traffic based on some set of rules. Usually these firewall rules are simplistic in nature and mainly work on source or destination of the network traffic. However, because of possible vulnerabilities in higher network protocols or applications, firewall rules usually need to have support for more complex functionality that for example take into account the network traffic amount, type or connection state.

This project aims at building upon an existing firewall solution in the Minix Operation System mainly with the intent to add such complex functionality in order to mitigate the affect of malicious or exploited user applications.

2 Vulnerabilities Countered

Most user applications today are downloaded and installed from the internet. While we as users most likely trust the source of our application, we can never be certain that malicious or exploitable aspects of the application do not exist. For example, imagine that we have downloaded a web client application such as the Firefox web browser. Instead of downloading and building the source code, we install a pre-built binary from a source that some attacker has gained access to. Once installed, the web browser opens and runs as expected. However, in the background it will perform malicious activities and attempt to either send information or allow remote access to the attacker.

There are techniques that could mitigate this form of attack but these techniques usually come with some drawback to either performance or usability of

the client application or system. One possible mitigation could be to run untrusted client applications within some virtual environment such as a virtual machine. While this approach most likely ensures the security of the clients actual device there could be noticeable affects on both performance and usability (i.e. no sensitive information may be added and used within the virtual environment without the risk of exposure). Furthermore, a valid approach might be to use existing firewall functionality with a white-list approach that only allows network traffic to and from trusted locations. The main drawback of such an approach is that such firewall rules might affect the usability of other applications that we already have established trust in. This project focuses on countering this vulnerability with the help of a more fine-grained control over firewall rules that allow monitoring and blocking network traffic to and from some untrusted application. We may see it as a form of network sand-boxing.

3 Design Choices

3.1 Accessing process information in firewall

One of the first design choices that had to be taken during this project was how to gain access to user process information in the firewall server. There are many ways to achieve this functionality however we have opted for a solution that we thought would be simplistic and not require any major changes to existing code. This solution works by utilizing existing socket creation functions that already include the user process endpoint ($endpoint\_t$). Also during the existing socket creation process, a Protocol Control Block (PCB) is created that later is used for both incoming and outgoing network traffic for that socket/process. To this PCB we chose to add the user endpoint which then allowed us to send that endpoint from the network stack to the firewall for every network packet that should be processed by the firewall.

3.2 Adding hooks

The firewall implementation from the last course iteration that we built our project upon opted to hook their firewall into the IP layer of the network stack. All network traffic in the system will pass through the IP layer of the stack and this layer mainly contains the packet type and source/destination IP address. While this information is sufficient for fundamental firewall rules, further parsing of higher level protocols in the packet must be done by the firewall in order to check firewall rules that work on this level (e.g. TCP/UDP/ICMP). If the packet passes firewall inspection this parsing will be repeated when the packet then is forwarded to the higher level protocol for processing. This redundant work in combination with the fact that the PCB is not accessible until this point was the reason why we have chosen to add firewall hook methods to most higher network layer protocols. Since these new hooks do not guarantee a one hundred percent coverage of all network traffic we have also redesigned the existing IP

layer hook to be more efficient and thereby allow it to act as a fallback hook to the firewall for each network packet.

3.3 Retrieving process name

The Inter-Process Communication (IPC) message sent to the firewall containing the packet for inspection will contain an endpoint that will identify the process linked to that packets destination or source. In order to link this integer endpoint to an actual application we designed an addition to the Process Manager Server (PM). Normally we would have to use the PM to query the process table and then locate the relevant endpoint, however we have added an additional SYSCALL to the PM that allows retrieving the application/process name directly using only the process endpoint. This gives a major advantage in performance and it is possible mainly because of the fact that process names have a 16 byte hard limit (which means it will fit in an 56 byte IPC message).

3.4 Modular decision servers

In order to limit the complexity of the main firewall server we decided to split up more advanced and stateful packet filtering to external server(s). For this project we were able to extract the TCP SYN-FLOOD protection algorithm from last years project into a minimalistic Minix server that the main firewall server easily can query for every TCP packet. This design allows the possibility of adding additional protocol servers in the future or changing or adding additional features to the TCP server with minimal or no code changes to the main firewall server and rules.

3.5 API for dynamically changing rules

In order to allow listing, adding and deleting firewall rules during runtime we designed and implemented and interface in the firewall server that can achieve this. This interface alone may be directly accessed by any other Minix server however for user level processes this interface is not accessible because of the server access model in Minix. A user level process is only allowed to send SYSCALLs to the Virtual File System (VFS) and the Process Manager (PM). Because we wanted a user level CLI to access the firewall we decided to add an exposing iterface though the VFS that is protected by root permissions. This implementation is fairly straight forward since the IPC message simply has to be proxied by the VFS to the firewall. Additionally we added a C functions that wrapped the firewall methods to allow easier usage by user programs.

```
// #include <sys/fwctl.h>
int fwdec_add_rule(uint8_t direction, uint8_t type, uint8_t priority,
                   uint8_t action, uint32_t ip_start, uint32_t ip_end,
                   uint16_t port, char* p_name);
int fwdec_delete_rule(uint8_t direction, uint8_t type, uint8_t priority,
                   uint8_t action, uint32_t ip_start, uint32_t ip_end,
                   uint16_t port, char* p_name);
```

4 System overview

The first main aspect of this firewall system is the process where the user endpoint is stored in the PCB. This is implemented by using the socket creation process described in Figure 1. When the user process wishes to open a network connection it will send a SYSCALL to the VFS asking it to create a socket with the PF_INET type. This type of socket is then handled by the alloc_socket function in the LWIP library module. This method already takes the user endpoint (for permission checks when creating RAW INET sockets) and then runs different socket creation functions depending on the protocol. Common to all socket creation functions is that they create the PCB that acts as a connection state container. Each PCB contains a set of attributes and a subset of these attributes are common for all IP based PCBs which allowed us to inject the user endpoint there.

Figure 1: Socket creation

Finally in Figure 2 we can view the traffic flow with regards to our implemented firewall.

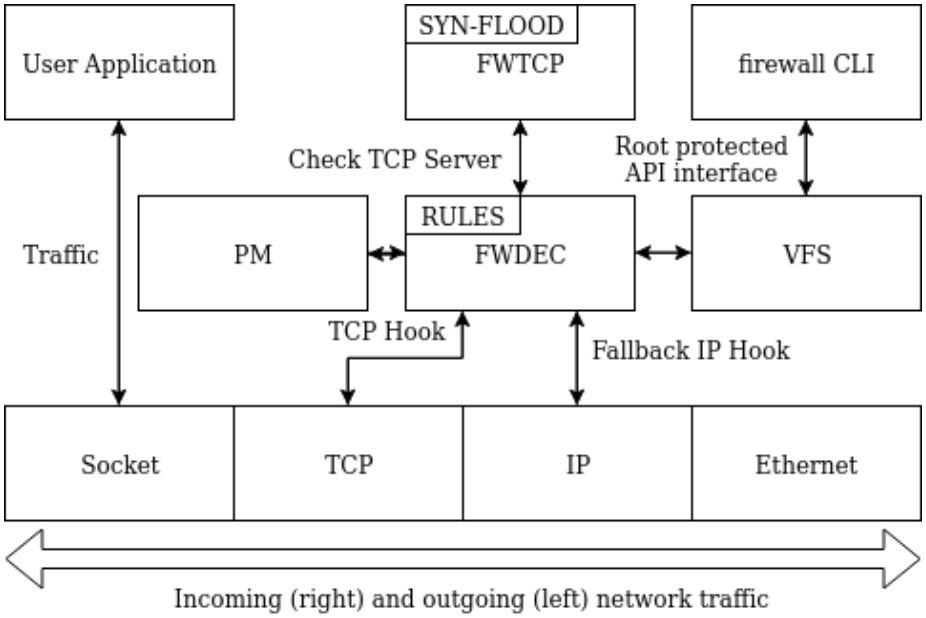

Figure 2: Overview of the firewall system

If we were to regard outgoing TCP traffic we see that the packets will first pass from the user process to the socket and then to the TCP layer of the network stack. The TCP function for outgoing traffic contains the PCB for the current packet so at this point the user process endpoint can be retrieved so that the hook to the firewall may be called. The main firewall server will receive the request to process an outgoing TCP packet. The firewall server will then call the Process Manager to retrieve the application name using the endpoint sent in the hook. Once the name is retrieved it will loop though all current firewall rules and look for the best matching rule for this packet. If the best matching does not block the packet a request is sent to the TCP server for SYN-FLOOD analysis.

5 How attacks are stopped

With our firewall solution the user is easily able to dynamically create firewall rules that limit the network capabilities of some user application. By using the command line firewall interface the user can for example start with creating a rule that blocks all network traffic to some untrusted application (e.g. Firefox). Then, the user may choose to run the application and monitor the logs for any attempts to send malicious traffic. To further investigate the application the user may also white-list certain locations and protocols to allow normal usage of the application while mitigating the risk of malicious traffic being sent.

6 My Contributions

Generally I have done a fair bit of refactoring such as moving firewall rule logic to a separate C header file and add support for all protocols and functionality of the rules. Excluding refactoring code from my project members during the course of the project there are a few noticeable additions I have contributed with to this project.

6.1 Adding endpoint to PCB

While this is a fairly minor change in the code it took many hours to figure out the inner workings of the LWIP library and discover how and where sockets are created and how network traffic propagates to and from them. After figuring out that the PCBs would be a suitable target for the process endpoint I had to find the location in the code to add this attribute. Once I found the generic IP PCB attributes the remaining code changes where easy.

6.2 Adding new firewall hooks to higher protocol layer

This change required changes to many different files since we decided to add hooks to TCP, UDP, ICMP as well as RAW traffic. The changes themselves are not too complex in nature however I had to put a lot of work into finding all

the suitable spots in the code to add the new hooks. This was difficult partly because we did not want to miss any traffic and partly because we wanted to piggyback on the parsing done on the protocol level (for example TCP SYN, ACK and FIN flags).

6.3 Move TCP related functions to modular server

While the commits related to this change contain a lot of code there is a large portion that I have simple re-added from the last years solution (mainly SYN-flood protection). What I did was essentially refactor the *myserver* bare-bone example server into a firewall TCP decision server that contains the SYN-flood checks that should be preformed on TCP traffic.

6.4 Create firewall CLI

One of my contributions that included a relatively large amount of code was the command line interface that allows the user to dynamically interact with the firewall. I designed this tool and its usage using some inspiration in existing Minix CLI tools such as *telnet*. In order to allow the CLI to access the firewall I had to replace the existing API method we had created with a method-/SYSCALL to the VFS that could do permission checks and then forward the message to the firewall itself. This was necessary since user processes are by default only allowed to access the VFS and PM servers in Minix. Also I wrote a full manual entry to the CLI tool.

7 Link to project

`https://gits-15.sys.kth.se/projsec19/minix-project/tree/johansson-v.1`