

System Design Document  
for  
CMSC 421 Project 3  
Phase: Preliminary Design  
Spencer R. Shimko  
2004-11-12

# Table Of Contents:

<b>1 Introduction.....</b>	<b>1-6</b>
1.1 Purpose.....	3
1.2 Scope.....	3-4
1.3 References.....	5
1.4 Overview.....	5
1.5 Constraints.....	5-6
<b>2 System Overview.....</b>	<b>6-7</b>
<b>3 System Architecture.....</b>	<b>7-11</b>
3.1 Architectural Design.....	7
3.2 Architectural Alternatives.....	8-10
3.3 Design Rationale.....	10-11
<b>4 Data Design.....</b>	<b>11-13</b>
4.1 Global Data Structures.....	11-13
4.2 Functional.....	13
<b>5 Human Interface Design.....</b>	<b>14</b>
<b>6 Testing.....</b>	<b>14</b>
<b>7 Timeline.....</b>	<b>14</b>

# **1 Introduction**

## **1.1 Purpose**

This document describes the implementation details surrounding the addition of a kernel level firewall to Linux. The firewall will filter incoming and outgoing packets based on ports specified at the transport layer of the OSI model. Additionally the firewall will facilitate filtering based on Internet Protocol (IP) addresses. The reader is expected to have a basic understanding of Linux kernel programming, Netfilter functionality (hooks), kernel data structures, and should at least be familiar with general network and communication concepts.

## **1.2 Scope**

The goal of this project is to provide a successful firewall implementation by using the framework provided by the Netfilter hooks. For this project to reach that goal several separate software components will be produced to provide a full range of functionality. This functionality will depend largely upon a key software component in the form of a loadable kernel module (LKM). This LKM will perform the task of filtering based on port policies and blacklisted IP addresses. There will be several sub-components that extend from this module to complement it's ability to filter. More detail regarding this LKM and Netfilter implementation will follow in later sections however a brief discussion to describe the metric upon which to view this firewall is appropriate.

A firewall's success is measured by a combination of it's effectiveness and it's adaptability. As such this system will need to be effective and adaptable.

A firewall's effectiveness is measured in it's ability to protect a

system(s) from an attacker. Whether this attack is intentional, accidental, from an external source, or internal source, is inconsequential. It's effectiveness can only be measured within the constraints of it's abilities. This firewall will only be examining packets for ports and IP addresses. It will be effective within these constraints. Other related material that will not be considered for this project are firewall related topics such as session states and malicious packet content (beyond that of invalid ports and IP addresses). As a result this firewall's effectiveness is not altered by it's failure to protect from situations stemming from these topics. The LKM will be responsible for effectively enforcing protection policies within the above described constraints. However this gives rise to another firewall metric, adaptability.

Artificial intelligence has not matured to a point where a firewall can self-adapt to changing situations. To continuously protect a system(s) from attacks an administrator's intervention may be required from time to time. Principally the administrator must alter the policies to meet the needs of an ever changing environment and then make the firewall aware of the new policy changes. In order for this module to be adaptable it must provide a mechanism for runtime configuration. Information must be provided to the administrator so informed policy decisions can be made.

Information will need to be provided to the administrator so informed decisions may be made regarding policy changes. This information will be provided through statistics provided by the LKM upon request by a administrator.

These components combined will provide for an effective, adaptive, and thus successful firewall implementation. Each of these components will be enumerated in the sections following.

## 1.3 References

### Project Description:

<http://www.csee.umbc.edu/courses/undergraduate/421/Fall04/project3-new.html>

### Linux Network Stack & Netfilter:

<http://www.phrack.org/show.php?p=61&a=13>

<http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html>

### Linux Kernel Module Programming:

<http://www.faqs.org/docs/kernel/>

### Kernel Source:

<http://lxr.linux.no/source/?v=2.2.19>

## 1.4 Overview

The remainder of this document will be dedicated to clarifying the statements made in section 1.2 above (Scope). Specifically, a more descriptive overview of each component of the software will be presented followed by a discussion of the architecture and data design. This document will then enumerate the details surrounding the human interface to the firewall including output of statistics and input of policies. A method for testing, exercising, and debugging the firewall will then be presented. Finally a brief time line for each stage of development will be laid out.

## 1.5 Constraints

Aside from the constraints placed upon this project from the description there are a few other restrictions to consider. Data must be copied safely from user space (the policy file) to kernel space (the linked list of ports). Additionally the packet inspection must be completed in a timely fashion to keep the packet queue at a reasonable size. The packet inspection code must be placed in the proper location in the network stack

to have access to the transport layer header. To ensure efficiency a packet should be dropped low in its traversal of the network stack. This prevents unnecessary processing of the packets that are going to be denied as a result of the policy.

## **2 System Overview**

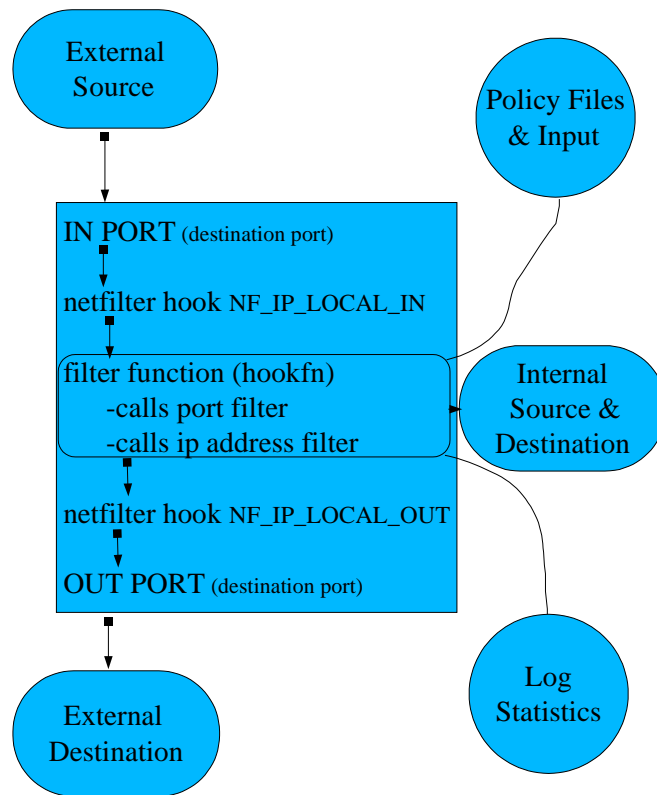
This firewall system exists in the context of the standard RedHat Enterprise Linux. A RedHat proprietary kernel version 2.4.21.EL-20 will be modified for this firewall system. While several alternative implementations were considered (see section 3.2) a loadable kernel module utilizing the Netfilter framework is the method that will be used for this project. The end result will be useful in a IP version 4 networking environment but could easily be extended to IP version 6 and other protocols. The goals of this implementation are described above but perhaps some background information regarding the subject of firewalls is appropriate.

Security is a important topic in todays world of input/output systems. With the ability to accept input of many forms computers are susceptible to malicious and accidental attacks whose results could be unwanted in the end system. Internetworked computers are of particular interest because they, by nature, leave must accept input of many types from external sources. Firewalls limit this input by denying certain types of traffic from entering the system. This can lessen the likelihood of an attacker successfully compromising a system and reduce the likelihood of an accident causing unwanted results. The firewall implemented in this system will protect a computer from attacks by denying input and output from specified ports. By reducing the number of open ports a user is reducing the number of entry points an attacker could exploit to gain

access to a system. A firewall user is also reducing the effect that an uninformed user has in network environment.

### 3 System Architecture

#### 3.1 Architectural Design



## 3.2 Architectural Alternatives

Hardware alternatives are numerous and thanks in large part to the cross- platform design of the Linux kernel this firewall will be able to run on over eighteen different types of hardware architectures. This system can be used on any hardware platform supported by the Linux kernel such as Intel x86, Sun Sparc, or Motorola PPC. This is due to the use of Netfilter for the implementation and the decision to use the proc filesystem for input and output. Because proc will be used for input the firewall will not require additional system calls. It is these system calls that typically cause cross- platform issues since they require modifications at a very low- lever to the code found in the arch/ sub- tree. Any modification of files residing in the arch/ tree must be considered carefully and careful steps must be taken to ensure cross- platform compatibility.

Software alternatives have been weighed carefully. Consideration was paid to the connect system call however the idea was quickly discarded as there is no simple method for determining the direction of packet flow at this level. Serious modifications would have to have been made at numerous spots in the kernel to track the flow of the packet at the connect system call level.

Additionally, the function `tcp_v4_rcv`, it's counterpart UDP receive, and their respective outgoing functions were analyzed carefully. While an implementation at this level could be quite functional it is far from efficient or maintainable. Even if two central filter functions were developed (for incoming and outgoing), code would have to be inserted within each network function to call the filter. This would quickly become a handful to maintain as protocols are added and removed from the stack. Additionally the code would be difficult to modularize since it would exist at numerous differing locations in the kernel.

Lower in the network stack functions such as `ip_rcv` (`ip_input.c`) and

`ip_build_and_send_pkt` (`ip_output.c`) appear to be valid insertion points. These looked promising because the direction of packet flow was indicated by the function itself (`ip_rcv` for input and `ip_build_and_send_pkt` for output). Additionally these functions provided a fairly central location for code modification and maintenance. The type of packet (transport layer for port analysis) could quickly be determined at this point by dereferencing a few pointers in the IP header. One drawback is that this implementation still required modification to the native kernel code unless IP was built as a module. If IP was built as a module then by including the firewall inside of the module effectively goes against the principals of modular programming. It is effectively combining two separate components that should be placed into logically separate components. However, analyzing these two functions had one major benefit. In the last few lines in each of these functions exists a call to Netfilter hooking function, `NF_HOOK`. This directly led to the decision to use Netfilter hooks for the implementation.

There were also several choices for statistical output and policy input. The most simple method would be to simply read the policy file on boot. This would restrict the adaptability and robustness of the implementation since policy changes could only be enforced after a reboot. An alternative was considered that added a system to handle the policy input. This was discarded for the reasons described above (system calls complicate portability).

Another method that could have been used was to create a device file in `/dev` and use this file to control the firewall. The benefits of this implementation include the ability to define `ioctl`'s for the device file and could make control from a user space driver program fairly intuitive. The drawback to this method is the safe creation of a device file. This would include choosing an unused device minor and major numbers in the

experimental range. This range is subject to change it would be difficult to ensure a unique entry on each machine. Additionally it would require administrator intervention to create the appropriate file using the "mknod" command and the correct major/minor numbers.

Considerable thought was put into ways to make the policy changes take place automatically. As soon as the file was altered the change would take place. Problems arise with this type of implementation with malformation of the policy files. How would the user be notified of such a situation? In the end it was the complexity of modifying the file system or file write functions to monitor a single file (too much overhead) that made this option too cumbersome to implement.

The overhead incurred by writing statistics to a text file or the kernel log with each packet is rather large. This led to the conclusion that perhaps there was a way to combine the stats and policy input in some logical way since they are both forms of exposing the user interface but only upon request. The proc filesystem was made for this purpose. One file for input and one file for output should allow this implementation to logically group the functionality while storing statistics without the cost of excessive overhead.

### 3.3 Design Rationale

Linux was chosen for this implementation for several reasons besides the requirements. It is extendable, open source, cross- platform, and popular. This made it the appealing for a project of this type. The hardware is limited only by the existence of a kernel port to the device and the existence of a network hardware.

Netfilter was chosen since it is highly extendable, modular, and is well documented. It has "hooks" in various parts of the network stack that allow a function to be called to analyze the packet at that point. This

function can make a decision to pass on, steal, or drop the packet. For the purpose of this implementation packets will only be dropped or passed on. The code can be written as a LKM and can easily be inserted and removed from a running kernel without having adverse effects on the entire system (other than the lack of security provided by the firewall module when it is removed). This makes the design and debug process significantly easier since the machine will not have to be rebooted to test the code and only the module's source will be compiled, not the entire kernel.

A new subdirectory and two new files will be created in the proc filesystem. One of these files will be used for statistical output and can be read like a typical read-only file. This will separate the logging from the packet filter. The filter will merely increment variables as necessary. The proc read function will be called only when the file is read and this function will calculate and present the statistics and a human readable form.

The second file added to the proc filesystem will be for input. It will allow the administrator to alter the policy file and have it read by the kernel by using that entry. At this time in the design process it is unknown whether the input will be the contents of the file, "cat /policy.txt > /proc/firewall/policy", or whether communicating with this file will cause the input function to read the file itself, "echo -n 1>/proc/firewall/policy". This decision will be made at implementation time.

## **4 Data Design**

### **4.1 Global Data Structures**

Several major data structures will be required for this system

implementation. The first is the two Netfilter structures that will be needed. One will be used to hook the incoming packets, the other will be used to hook the outgoing packets. Netfilter hook structures are fairly well defined and once declared they must be filled with certain types of information that define where and how the work works.

Next, three kernel linked lists will be needed. This data structure was chosen to ease implementation and because it satisfies the necessary requirements (iteration, kernel space storage, and it is a dynamic structure). The first two linked list will represent the data from the policy file regarding ports. One will be used for IN port specifications and the other will be used for the OUT specifications. This is the logical place to perform logging as well since statistics will need to be reported for each port. To this end a structure will be developed that will store a short integer, storing the port, and an integer storing the number of "hits" on that port. The linked list will store elements of that structure type. The third linked list will be very similar and will store the IP addresses specified by the blacklist. The size of the structure's elements will differ however. It will need a integer to store the IP address and a integer to store the port. Note that since the statistical information will be monotonically increasing a check will be performed to ensure the valid size of an integer is not exceeded. A decision on how to handle this situation will be made at a later time.

A few additional variables may be added to enhance the statistics but most of the data can be calculated as necessary from the above structure.

A structure will be defined for each entry in the proc filesystem. Once again, the setup of these structures is fairly well defined. The choices involved in setting them up are crucial, but limited. One will be enabled for reading, while the other will be enabled for writing, an effort

to keep things logically separated.

Some other global data structures will be used that are provided by the existing kernel Netfilter code. For example, the linked list of Netfilter functions at a specified hook will be utilized but its functionality is hidden by calls to other functions.

## 4.2 Functional

```
unsigned int hook_function(unsigned int hooknum, struct sk_buff **skb,  
    const struct net_device *in, const struct net_device *out,  
    int (*okfn)(struct sk_buff *))
```

There will be two hook functions. One will be used for the input hook. The other will be used for the output hook. This function is "hooked" into the Netfilter chain and will perform the filtering. The first parameter specifies the hook type such as: `NF_IP_PRE_ROUTING`, `NF_IP_FORWARD`, `NF_IP_POST_ROUTING`, etc. The second parameter is a socket buffer (representative of a packet). It is this buffer that holds the information that will be used by the firewall such as ports and IP addresses. The next two structures are used to define the interface a packet is entering and/or leaving on. For this firewall implementation netdevice "in" will only be defined for the input function and the "out" will be NULL. For the output function that will be reversed.

```
int read_func(char* page, char** start, off_t off, int count, int* eof,  
    void* data);
```

This will be the callback function for the proc filesystem read entry. The first parameter is the buffer to fill for output. `start`, `off`, `count`, and `eof` are used to more accurately simulate a file but will not be significant for this implementation. `data` will not be used in this as well.

```
int write_func(struct file* file, const char* buffer, unsigned long  
    count, void* data);
```

This will be the callback function for the proc filesystem write entry. The first parameter, `file`, will not be used. The buffer contains the user data to pass to the kernel. This buffer exists in user space so it must be handled using the `copy_from_user` function. The `count` contains the count of bytes in the buffer. `data` will be ignored in this implementation.

## **5 Human Interface Design**

The user will interface with this firewall using the proc filesystem. The user informs the kernel of policy changes through the writable proc entry. It reports statistics through the readable entry. Another mechanism for interfacing may be the module itself. Since it can be loaded and unloaded at runtime, disabling and enabling the filter, this is another method for a user to control the firewall at a much higher level.

## **6 Testing**

This system will be thoroughly tested by exercising its full capabilities. An extension will be added to the policies that will allow the firewall's default policies to be switched from drop, to accept. Local IP addresses (127.0.0.1) will be tested as well as remote incoming connections and outbound connections. The statistics will regularly be analyzed in a controlled environment to ensure they are being accurately updated and reflect the correct information.

## **7 Time line**

Released: October 25th, 2004

Design complete: November 12th, 2004

Implementation complete: December 1st, 2004

Debug and fine tuning: December 10th, 2004

Final Submission: December 14th, 2004