# Firewalls: Programming and Application

An Honors Thesis submitted by

Terry Rogers
258 Providence Road
Telford, TN 37690
(865) 437-9798

A BS in Computer Science

April 27, 2011

Project Advisor: Dr. Henry Suters

## Dedication

This project is dedicated to the memory of my mother, Linda Lockhart Rogers Freeman (1960-2010). Regardless of how difficult her life was, she always made time for her son and made his education her top priority. She will always be loved, remembered, and dearly missed.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# An Introduction to Firewalls

The purpose of this project was to design a firewall program to defend a computer from malicious attacks. Firewalls are important tools and are often the first line of defense for a computer. The firewall was designed, tested, and attacked by myself using the knowledge I gained from researching this topic. Through this one can gain a better understanding of security and networking practices from both defending and attacking perspectives. The importance of information security also comes to light through this project.

Information security has been a concern throughout history. Even in ancient times a desire existed to keep certain information confidential, accomplished through the use of methods such as the Caesar Cipher that was reportedly used by Julius Caesar himself [20]. In more recent times, the German Navy used a code machine during World War Two, named Enigma, to keep information such as ship deployments concealed. When the code was broken, everything the Germans transmitted was available to the Allies, aiding in the protection of convoys

in the Atlantic and eventually their victory in the war [7]. As computers became widespread, a way of securing the computers as well as the data stored on them from malicious actions was needed. Without security, vital services and untold millions of dollars can be lost, and the number of attacks is increasing [19].

A firewall is an important tool used in securing computer networks. There are 4 different styles of firewalls, along with different means of implementation, but for the purpose of this paper a *firewall* will be defined as a software program dedicated to inspecting all data that flows between a computer and the network to which it is connected [17]. The firewall acts as a filter, examining traffic that comes in and comparing it to a set of rules. The firewall then decides whether to forward the traffic or to "drop" it, preventing it from ever reaching its target destination.

## 1.1 Networking Terminology

In order to understand the firewall, a few basic networking terms need to be clearly defined in order to understand what each firewall style does and how it differs from the others.

### 1.1.1 Packets

The majority of information on computers, including that sent across networks such as the Internet, is sent in packets. Packets are blocks of data: the computer sends out the data in these blocks rather than in a constant stream. Each packet contains a large amount of metadata along with the data itself. In packets sent

along a network this metadata includes the source and destination IP address and port numbers, the total number of packets, flags to denote the type of data carried, and more.

Each packet bounces from computer to computer in a network until it reaches its destination or times out. The route the packet takes is dynamic, an intentional feature designed when the research network ARPANET was first created to share scientific information and resources. ARPANET's design first gained momentum shortly after the *Sputnik* launches by the Soviets [5]. Since the only computers that were hooked up to the network were all authorized research computers, security was not a major issue outside of physical security, designed to prevent damage against physical attacks and espionage. By the time the network expanded and security became important, too many machines depended on it to change the design to make it more secure. The ARPANET design eventually evolved into what is known as the Internet today, carrying both its strengths and weaknesses to its new form. Due to these protocols being used in an unforeseen manner from the time of their initial conception, security products such as firewalls are needed to protect computers from malicious attacks.

### 1.1.2 Protocols

Protocols are rules that govern how two devices communicate with each other. There are dozens of different protocols for different devices, whether each is a cellphone, printer, or computer, but all styles of communication have at least one. Protocols can be thought of as the languages of the devices: both devices must be using the same protocol (language) in order to communicate. If they do not, no

meaningful communication is possible. The protocol used depends on the needs of the situation, and often new protocols are created to suit specific devices.

There are two specific protocols referenced continuously in this paper: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). There are many differences between the two, but the main difference is that TCP focuses on stability and reliability whereas UDP focuses on speed. TCP constantly sends confirmation of packet deliveries and requests for lost packets while UDP does not send or receive any confirmation of packet delivery. TCP is more common than UDP though both are used frequently in day-to-day information transfers. For instance, TCP is the main protocol used for web pages and e-mails while UDP is commonly used for real-time video and audio streams.

### 1.1.3 Internet Protocol (IP) Addresses

An IP address is a set of numbers that indicates where an object on the network is. As long as a packet has the correct IP address and a link to the destination, it can reach the target computer. IP addresses are usually handled either by a network policy with an administrator assigning values or automatically assigned to a computer when it connects to a network.

Most IP addresses are 32 bit, referred to as "IPv4"; the IP is split into four sets of numbers ranging from 0 to 255, called octets, with each set separated by periods. For example, 230.105.4.32 is an IPv4 address. Due to the rapidly diminishing number of IPv4 addresses available as more and more computerized devices connect to the Internet, a new address format, dubbed "IPv6", has recently begun to see use, though it is not yet commonplace enough for the concerns of

this project. This version uses a 128 bit number, which allows a far wider range of possible addresses. Each packet specifies both the IP it initially came from, which is referred to as the source IP address, and the IP address that it is trying to reach, known as the destination IP address.

### 1.1.4 Port Numbers

Each packet also specifies its originating and destination ports. Ports are used as a way to identify what the packet is used for and to forward it to the desired process. There are over 65,000 ports available though the majority are unused on a daily basis. Certain functions such as HTTP for web browsers have an informally agreed upon port. While the port can be changed if so desired, these defaults simplify the connection for users. Port numbers are usually added to the end of an IP Address and separated by a colon. For example, a packet that is destined for IP address 230.105.4.32 and port 80 would have a destination address of 230.105.4.32:*80*.

Programs can "listen" on a port so that any incoming traffic that has that port number listed is available to the program. However, only one program can listen to a port at any specific time. Because of this, in a basic setup only one web browser could be connected to the Internet at a time from a particular IP address. Most operating systems circumvent this limitation by an intermediate process that sifts through all the packets that come to a port and sends them out to the appropriate application. This allows for a user to run multiple programs using the same port, such as Mozilla Firefox and Microsoft Internet Explorer, simultaneously. A postal worker would be a good example of this process. He

or she may receive multiple letters designated to the same street address, the IP address, which leads him or her to an apartment complex. From there the worker would deliver the letters based on the apartment, or port, number.

### 1.1.5 Subnet Masks

Subnet masks break the IP address up into two groups: the network prefix and the host identifier. Subnet masks use the same formatting as a full IP address, utilizing four numbers ranging from 0 to 255. The look of the IP address and the subnet mask differ in the bit pattern. The subnet mask must contain consecutive ones. Once a zero is reached, all following bits must be zero as well. The bit patterns are then matched up to the IP address, and the portion of the address that matches to the ones of the subnet mask are designated as the network prefix, denoting to which subnetwork the packet is destined. The rest of the address then designates which computer within the subnetwork is the exact target.

| | Subnet Mask | Bit Pattern |
|---|---|---|
| Valid | 255.255.255.128 | 11111111.11111111.11111111.10000000 |
| Invalid | 255.255.255.64 | 11111111.11111111.11111111.01000000 |

**Table 1.1: A valid and invalid subnet mask**

Subnet masks are often represented by a forward slash (/) after the IP address, followed by the number of ones in the bit pattern. For example, the IP address 230.105.4.32 and the subnet 255.255.255.128 can be represented as 230.105.4.32/*25*. Because of the way subnet masks restrict the number of bits in the IP address for the host identifier, subnet masks can indicate how many possible hosts are available on a network. For instance, a subnet mask of 255.255.255.0

indicates that there are a maximum of 256 available hosts for the subnetwork. Subnet masks also allows the same IP address to be used for different computers; the addresses 230.105.4.32/20 and 230.105.4.32/24, while being the same IP address, indicate two different subnetworks and thus two different target computers.

## 1.2 Firewall Categories

Almost every firewall falls under one of four type categories, each one having specific advantages and drawbacks. Each style uses a different method of protection, though the main purpose stays the same: each follows a specific set of rules to filter out incoming and outgoing traffic to protect the network on which it is installed. Which one is used depends on the user's requirements of the firewall and the resources available.

### 1.2.1 Packet Filters

Packet filters use a specific set of rules configured to examine each individual packet's source and destination information, usually the IP addresses and port numbers. If the packet falls under one of the rules, the rule contains an instruction to either drop or allow the packet. If the packet does not meet the specifications of a rule, then the firewall falls back to a global policy. The global policy is set to either drop or allow all packets that do not have an applicable rule in the ruleset. Most firewalls give the user the choice to pick his or her own rules and global policy, though some may come with pre-configured rules. An example of this would be a firewall with a rule allowing all packets that contain a source IP

of 230.105.4.32 and port number 25, while the global policy states that all other packets must be dropped. Any packet that does not show a source IP and port of 230.105.4.32:25 would be instantly denied access and dropped, cutting it off from its destination.

This type of firewall is very useful to create either a whitelist, to allow only certain computers to connect on a private network, or a blacklist, to block known malicious IPs. However, a trusted source can be taken over and controlled by others. It is even possible to spoof IP addresses in packets to make them appear to come from a different location. If this happens, the firewall will continue forwarding the packets and become useless [1]. Packet filters are relatively quick though, so they do not significantly impact the network or computer speeds [17].

### 1.2.2 Application-level Gateways

Application-level gateways also filter packets, but do so based on the application data within the packet. This way, the firewall can block specific types of services, such as all Java applets within a web page [18]. It also offers the protection of a proxy on the network. Normally when a connection is established, the two computers communicate directly with each other. With a proxy, each computer sends its data directly to the firewall itself, but the computers still believe that they are sending information directly to each other. In other words, the firewall acts as a middleman between the two parties. It even has its own IP address separate from either computer. The main benefit of a proxy is that the IPs behind the proxy are hidden to outside sources. Because of this camouflage, the firewall must be attacked and broken before the computers behind the proxy can be specifically

targeted [6].

Application gateways offer more protection than packet filters do. They do not require specific source and destination addresses. Because they scan the application data of the packet rather than the source and destination information, they are more effective at protecting computers from potentially damaging services. Rather than attempting to block every IP that may be harmful or blocking a whole port to prevent abuse, such as port 80 for HTTP, the firewall can block specific actions taken by those services while allowing legitimate use. However, because it does examine the entire application data of the packet, it can severely slow down network traffic and use a lot of processor time. Also, many applications must be manually customized to deal with the proxy service of the firewall [17]. Because the application expects to talk directly to the other computer, any interruption in this conversation can cause errors or trip built-in safeguards.

### 1.2.3 Circuit-level Gateways

Circuit-level gateways function similarly to application gateways in that they too use a proxy to secure the network. When the circuit gateway scans a packet, however, it often only checks that the connection is legitimate, insuring that the incoming packets are coming from the same source that the request was sent to and that the user has sufficient privileges for whatever service is needed [17]. Once the connection is established, the firewall only checks if the packet is part of an existing connection. If it is, the packet is then passed on without any further inspection or interference.

Circuit-level gateways offer a wider variety of possible rules than other firewalls

along with its proxy service, so it is more versatile in this regard. For example, it can block specific URLs from being accessed, whereas a packet filter would need the IP address of the website, which may not be unique since many large companies such as Google use multiple servers. Also, circuit-level gateways do not significantly slow down the network or computer, similar to packet filters in speed. However, most circuit-level gateways will only check certain flags of the packet to identify if it is part of a connection rather than examining a list of previous connections. These flags can easily be modified to fool the firewall, which makes this type highly vulnerable. In addition, previously legitimate connections that get approved can quickly turn malicious, yet the firewall will not intervene in an existing connection. As such, this type of firewall is rarely used by itself to secure a network [17].

### 1.2.4 Stateful Packet Inspection (SPI)

SPI firewalls are a hybrid type combining features from each of the three previous firewall styles. Most SPI firewalls function according to the following steps. First, the firewall examines if the packet is part of an existing connection, similar to circuit gateways. If it is not, the firewall then sees if the packet is allowed by a set of rules that resemble a packet filter's ruleset. If the packet does not fall under either of these procedures, then the packet is dropped. If the packet passes either of the previous tests, the packet's data is then scanned in a manner similar to application gateways. If it violates any rules, it will then be dropped. If the packet passes this step, it will finally be forwarded to its destination.

The proxy feature of application and circuit-level gateways is sacrificed on SPI

firewalls for the sake of speed and simplicity. Also, the data of the packet is only scanned for specific strings rather than examining the packet in its entirety, which is what an application-level gateway would do, trading security for speed once again [17]. However, the procedure to check if the packet is part of an existing connection is improved. Instead of only checking flags, the firewall checks the source and destination IP addresses against a list of currently existing connections that it maintains. This makes bypassing SPI firewalls much more difficult, as an attacker cannot initiate an attack without the target requesting or accepting the connection first. While SPI firewalls are slower than packet filters, they are faster than application gateways. SPI firewalls generally seek the middle ground in terms of security and efficiency.

## 1.3 Computer Crime Laws

Laws concerning computer crimes had to be looked at before the firewall could be fully developed and tested. These laws are not always clear and constantly change to keep pace with the rapidly evolving technology they cover [16]. The majority of laws are made at the state or local level, with only a few broader laws at the federal level. As such, legality varies from location to location. Anyone who does any kind of security testing must know the law since even testing one's own networks could be enough for one to be fined or imprisoned.

For instance, in some states it is illegal to be in possession of lock-picking tools, and even then the punishment can vary from state to state. The same is true for digital "lock-picking tools" such as port scanners and encryption crackers. Merely

having such software on a computer can lead to being charged with a crime in some states, with the punishments also varying in each location [16]. Many companies have naively asked a security tester to perform operations that are illegal in their area without knowing so, leaving him or her fully responsible for any damages and illegal acts that occur. As such, anyone who deals with computer security must know and understand local, state, and federal laws regarding computer crime. In the end, the best way to protect oneself from infringement is to both know the laws and, in the case of working for others, to sign a contract absolving oneself from any possible charges from the person or company that one is working for.

For the state of Tennessee, activities such as port scanning were completely legal at the time of this project. Any intrusion done on a computer, such as in the enumeration process (defined in Section 2.1.2), was also legal at the time as long as there was no *malicious* intent behind it and that the person doing so had proper authorization [8]. This project used personal devices with no malicious intent, thus all actions taken and tools used during this project were done in a legal manner.

## 1.4 Goals

For this project, I have designed an open-source firewall for the Windows platform. The firewall is able to filter packets based on the IP address metadata and the application it is designated for. This, combined with Windows Firewall, which employs a circuit-level based inspection, implements a SPI firewall-styled security system.

Along with designing the firewall, I also tested it by attempting to use port scanning and enumeration tools. Not only does this insure that my firewall is working properly, it also gave me more experience and insight into the opposing side of the battlefield. Ethical hackers throughout the business world constantly test the security of companies to find any weak points before someone with malicious intent does and costs the businesses untold amounts of money [16].

Now that the goals, laws, and basics of firewall designs have been discussed, the next chapter will focus on the specifics of this implementation, including more in-depth terminology and concepts, in order to bring a better understanding of this project.

# Chapter 2

# Essential Tools and Definitions

## 2.1 Advanced Terminology

Before discussing the inner workings of the firewall design, more in-depth terms must be defined concerning both security testing and programming in general. While some are common enough to not require any detailed definition, others may be less well-known and understood.

### 2.1.1 Port Scanning

Port scanning is a method designed to discover whether a computer is connected to the network and running. If so, a port scan can identify which services are available on the computer [16]. It does this by sending packets with different flags set to see what type of response the target computer gives and if the response gives any indication as to the defenses on that port. As with several implementations involving computers, there was and still is a legitimate use for the concept, but it

has evolved into one of the main methods attackers use to probe for weaknesses in a target computer. There are many different types of port scans. Each type is defined by the combination of flags that are set on the packet that is sent to the target.

The port scanner returns one of three different values for the ports it tests: open, closed, or filtered. "Open" indicates that the service is available and can be used without major interference, whereas a "closed" port means the service is not running and no connections will be accepted. A "filtered" return indicates the port may have a firewall watching it, but it is not conclusive and the port may in fact be better labeled as open. Port scanners have evolved since their initial designs; they also often include a best guess based on the type of returns it gets on what type of operating system is running, another piece of information that can be used by an attacker.

### 2.1.2 Enumeration

Enumeration is much like port scanning but with the intent of discovering more useful information from the targeted computer. Using enumeration, the attacker can find information such as what resources are on the network, what shared items exist, and even user names and passwords [16]. Unlike port scanning, enumeration is operating system specific: the processes and tools used to enumerate, for example, a Linux and a Windows computer, differ. Because enumeration digs deeper into the target computer than port scanning, its use is more heavily restricted. Though some states may allow port scanning without consent of the target, others may determine that regardless of circumstance enumeration is too intrusive and

thus is an illegal act.

### 2.1.3 Application Programming Interface (API)

Application Programming Interfaces (referred to as APIs hereafter) are common in programming. APIs allow the programmer to use functions in a simple manner through the use of the interface functions. By designing the functions this way, the programmer does not have or need access to the specifics on how exactly the functions are implemented. The programmer need only know how to use the API functions in order to successfully implement any method that is provided by the API. For this project I referred to and used several APIs, including the Packet Filtering API, Windows Firewall API, and others.

## 2.2 Tools

Different types of tools, namely software, were used in this project. The following overviews show a few of the major ones that are referenced throughout this paper and explains why and how it was used.

### 2.2.1 Visual Studio 2008

For my compiler, I used Microsoft's Visual Studio 2008. Visual Studio has two advantages over other potential compilers. The first is that it compiles Visual C++, which is a variant of C++ created by Microsoft and was the language used to code my project. It has the functionality of C++ with additional features, most of which are specific to Windows machines. By using Visual C++, a workable

form of my project was created in an easier and more efficient manner, allowing me to interact with needed functions stored within Microsoft DLL files. The other reason Visual Studio was chosen was due to its effective and simple debugging process. That ability aided me in quickly identifying and correcting any errors within my code. The functionality and importance of the debugging mode will be explained further in the next chapter.

### 2.2.2 Open Source Firewalls

I also used multiple open source firewalls, namely Firewall PAPI, NetDefender 1.5, and `Packet_Filter`, as references. The Packet Filtering API, used to block packets via IP address and port data, has been discontinued by Microsoft for Vista and beyond, and as such documentation on the commands and process is scarce. Thus Firewall PAPI was most useful in this regard, though not nearly as efficient as it would have been if proper documentation was available. NetDefender 1.5 takes a different approach and does not use the Packet Filtering API, but it also includes features not found in Firewall PAPI, such as a port scanner and a list of current applications and the ports they are using. NetDefender proved useful in providing insight into the workings of the API due to its similar formatting and, because of its usage of a different method, also provided a possible alternate to the Packet Filtering API used in Firewall PAPI.

During the process of creating this firewall, I was forced to switch to the Windows 7 operating system. The reasons behind this will be discussed in detail in the next chapter, but for my purposes the `Packet_Filter` firewall was very useful. It uses the Windows Filtering Platform and provides a simple example of

how the functions of the platform work together. Though the Windows Filtering Platform has ample documentation available, a reference of how it is used is still beneficial, as the documentation only describes what each function does and not how to combine them into a working program.

### 2.2.3 Nmap

Nmap is a port scanning tool originally designed in 1997 [16]. It allows almost anyone with basic knowledge of port scanning to perform a scan on any computer as long as the user has the target's IP address and a connection to it. Its size, simplicity, and effectiveness are the main reasons that it is such a popular tool, popular enough to even be used in the movie *The Matrix Reloaded*. It was used to test the firewall and examine the type of data that could be retrieved in order to determine the firewall's effectiveness.

### 2.2.4 NetScanTools Pro 2010

NetScanTools Pro is a multi-feature scanning tool, including options for port scanning and enumeration of Windows platforms. For this project it was used mainly in its enumeration role in a manner similar to Nmap. While NetScanTools is intended to be used on one's own machines to detect vulnerabilities, it is just as often if not more so used to find vulnerabilities in other machines for malicious purposes. For this reason it may be illegal to have it on one's possession in some states, though this does not apply in Tennessee. The use of NetScanTools is so regulated that I had to send a request form to the company and wait two weeks for it to be approved before I could download and use a simplified demonstration

version. Each of the more intrusive scans also display messages before they are run, warning the user that the actions could be deemed illegal if done without permission on the target computer.

## 2.3 Final Goal

My initial goal was to create a firewall using the Packet Filtering API and Windows Firewall API methods for the Windows XP SP2 platform. However, due to complications and other changes that will be discussed in detail in Section 3.3, I was forced to change my design to use the Windows Filtering Platform implementation for Windows 7. However, the final goal remained the same: to produce a working SPI firewall for the Windows platform, the process of which is discussed in the next chapter.

# Chapter 3

# The Firewall Creation Process

The process of creating and designing a firewall is not a simple one. One major decision that must be made is in regards to which operating system it will be designed for. The operating system is chosen based not only on its capabilities but also its usage as well. A firewall does little good if it only works on a system that no one uses. Once the operating system is determined, the method of implementation must also be decided. A balance should be found in regards to the complexity and effectiveness of each method.

## 3.1 Windows XP and the Packet Filtering API

Windows XP was the initial operating system of choice based on numerous factors. A Windows-based system was chosen over others, such as Macintosh OS X, due to the lack of open-source firewalls that are readily available for Windows systems. The majority of open-source firewalls available are designed for some form of Unix or Linux. My firewall is thus a much more original creation and can contribute

towards the field of security in regards to Windows-based systems. Windows itself is also far more popular than any other OS, giving the project a wider audience.

At the initial time of selecting an operating system, October 2009, Windows XP was the major operating system in use. Studies indicate that anywhere from 58-70% of the market share belonged to Windows XP. They also show that the use of Windows Vista was not growing, indicating that the majority of Windows users were more inclined to stay with Windows XP than to upgrade [11] [13] [14]. In terms of available resources for this project, both the Carson-Newman computer labs and my personal machine were running Windows XP at the time, so I did not truly have an option to pursue a firewall for a different version of Windows, such as Windows Vista. As such, Windows XP was the most suitable operating system for this project.

The Packet Filtering API method was chosen due to the availability of an open-source implementation, Firewall PAPI, which could be used as a reference and guide for the creation of my own firewall. All methods of creating a personal firewall for Windows XP, including this API method, lack proper documentation and support for both creation and usage. The lack of documentation was due to the fact that the API was available for a relatively short amount of time and was rarely used, as most software vendors chose to use a different method to access the functions directly, methods that date back to early Windows versions. It is also possible that Microsoft removed the documentation to promote and encourage development on the Vista platform. Because of this, open-source programs were the primary guide in learning how to interact with the system and create a functional firewall. The majority of the other implementations did not have any readily

available open-source material, and those that did lacked proper documentation necessary to ascertain the abilities and methods involved.

The Packet Filtering API was implemented in the Windows NT operating systems as a way for programmers to interact with the network traffic and information that Windows, along with the majority of other operating systems, attempts to hide from normal users. The specific function of this API is to give the programmer the ability to filter packets based on IP and port information; these actions are not available through the Windows Firewall interface. Thus the Packet Filtering API is the only method available to filter packets that does not require extensive knowledge of the structure of Windows NT operating systems, which is beyond the scope of this project.

### 3.1.1 Firewall PAPI Examination

I first examined Firewall PAPI in order to identify the key functions required to interact with the Firewall API. The majority of the code was irrelevant and pertained only to the user interface, however I soon found that the core of the API relies on the `PfAddFiltersToInterface` command. Through this function, a `PF_FILTER_DESCRIPTOR` structure that contains all the necessary information about what IP to block is passed, giving the API the information it needs to successfully implement the ruleset. This information includes the protocol and the destination and source IP addresses, along with port ranges. The filter is then added to the interface based on the direction of traffic that is chosen, either outgoing or incoming.

The interface was created by passing it the local IP of the adapter that it

is attaching to along with the default action: what the firewall is supposed to do if a packet does not meet any of the rule criteria. All the interface handles were then stored into a hash table, which allows the program to efficiently store and retrieve information, both for later reference and to be able to match which interface belonged to which IP address. By cross-checking the IP addresses with the interfaces, it is insured that multiple interfaces are not created and reduces the possibility for errors.

Overall, Firewall PAPI was a complicated program created to perform a simple function. The majority of the code was not commented, which is crucial for outsiders, even fellow programmers, to understand the process used. As previously mentioned, most of the code was also dedicated purely to creating and maintaining the graphical user interface (GUI), along with some code that seemed to have no noticeable function at all. The project file, which has metadata necessary for the compilation of the program, was corrupted and would not load. Because of this I could not independently compile and test Firewall PAPI. The lack of documentation, both inside the program and outside, made reverse engineering the code in my attempt to understand the API functions far more difficult than it should have been. What should have taken days instead took weeks.

Besides time, two major problems were discovered during my time examining Firewall PAPI. The first problem was the lack of customization with the Packet Filtering API. Based on the code and documentation provided, the firewall could only be used to block connections based on very specific IP addresses and ports. As such, it could not be expanded to look at the packet data in greater detail. This led to the larger problem, which is the blind faith that is needed to trust the

API. Since the functions do not appear to have the ability to examine packet data outside of source and destination information, one cannot be confident about the level of access the API has. If this is all the data that is available to it, then there is likely a middle-man function that it too relies on that does the packet examination and relays only the address information. If this is true, then the firewall's link to the traffic data is weak and the other function may be a possible security hole. Due to the circumstances of my available resources, I was forced to use this method regardless of these concerns and trust that nothing was intercepting or misinterpreting the data the API receives, else the firewall would become ineffective.

### 3.1.2 Firewall API Implementation

The initial phases of the implementation was plagued with errors and problems. Certain options, such as threading and Common Language Runtime support, had to be set to a specific setting in order for the program to compile at all. Because the Firewall PAPI project file was corrupted, I had to find and manage these settings myself. Other options and variables, such as the Windows Version, had to be manually set as well in order for the program to properly execute.

Once these were set and verified, I began implementing the very basics of the firewall. Using Firewall PAPI as a guide, I was able to quickly implement a simple program. This program had a very basic user interface that had the ability to start and stop the firewall as well as showing whether the firewall was currently running or not. All the needed data, such as the protocols and IP addresses, were coded statically into the program rather than accepting a dynamic user input in

order to simplify initial testing and minimize the possibility of errors. The only code that was implemented that was not a function of the Packet Filtering API or part of the interface was a method to convert IP addresses from string values into a format that was readable by the API.

On my first attempt at compiling this firewall I encountered an error that I could not correct. The type of program I was using and the function calls I was making, namely those relating to the Microsoft Foundation Class Library that allowed me to access the Windows API, would not compile properly under a debug version. The exact reason for this was unknown, and solutions ranged from changing settings to removing certain libraries from my program [2]. It would, however, compile and run properly as a release candidate. Because I could still compile properly in another version and the lack of information available about this error, I decided to manually debug the program using the release candidate instead of trying to fix the debug version. This severely complicated the debugging process, as I could not monitor variables and error codes or create breakpoints in the code as the program was running, features which a debug version provides. However a debug version is not essential to the completion of a program, as it can be debugged just as effectively through a release candidate, albeit with less flexibility.

After this difficulty was bypassed, the first test of the firewall was far more successful than my initial expectations. With the proper values for the protocol, TCP, and an IP address, 0 to indicate all IP addresses, I was able to successfully block all TCP/IP traffic on my computer. However, I soon learned that I was unable to block any traffic by using a specific IP or port value. Assuming this to

be a problem related to my code, I began a rigorous debugging process in order to determine the cause.

### 3.1.3 Firewall API Difficulties

My debugging process was very extensive and covered all possible areas. Because I was able to block all the traffic, I assumed that the firewall methods themselves were functioning properly and that it was an error in the values I was passing it. I initially thought the blame might lay in my method of converting the IP address strings, as this was the most likely place for an error to occur. However, even after I corrected a mathematical error in the function I still could not filter packets based on a specific IP address. I then looked at the subnet mask, as a generic one was used for the addresses I was testing and it could be an incorrect value. However, changing the subnet mask value only caused my firewall to not block any traffic at all. Even when I attempted to block all traffic, which previously worked, the traffic bypassed the firewall. Thus I knew that the value I was using for the subnet mask initially was the correct one.

After some time I decided to check Firewall PAPI and insure that it still blocked IPs correctly. I found it to have the exact same issues as my firewall did: even though it could block all traffic with the proper input and put the computer into lock-down, it could not block a specific IP address and allow others. I researched the project's development page and other coding forums, but I saw no complaint or mention regarding this obvious flaw. The only mention of my difficulty came after one user recommended Firewall PAPI to another, yielding the same blocking difficulty. From this I concluded, since such a bug would destroy the

entire purpose of using a packet filtering firewall, that at one point Firewall PAPI was able to filter packets based on IP addresses properly. Obviously something unrelated to the firewall had since changed.

Since it worked in the past, I decided to verify that I was using the proper operating system and if any updates had, either intentionally or as a side effect, disabled or corrupted part of the API. The Packet Filtering API was rated to work properly on Windows XP SP2, and since I was running Windows XP SP3 I made the decision to downgrade my computer. Even though SP3 made no official changes to the Windows Firewall, which the API partially relies on, according to the release notes, I knew from previous experience that any update can cause a number of unintended side effects [9]. My downgrade did not affect my firewall or Firewall PAPI in anyway, and as such I had run out of options, as determining the exact combination of Windows XP updates would be an impossible task. These difficulties, combined with my previous concerns of the weaknesses of this method, forced me to conclude that I must abandon the Packet Filtering API implementation method and pursue an alternate one.

## 3.2 NetDefender 1.5

Initially, I researched NetDefender 1.5 as a possible source to continue my firewall using Windows XP. NetDefender uses an approach similar to Firewall PAPI in filtering packets in that it too is a packet filterer. NetDefender also uses a form similar to Firewall PAPI for its functions and formats the data it requires in the same manner before using it in the firewall methods. However, where PAPI

uses the Firewall API, NetDefender uses a hook to access functions directly from Windows' DLL files.

The reasons I did not use this approach initially were twofold. First, although this method was more directly involved with the functions than using an API, I had no way to identify what these functions actually do: I would essentially be forced to use them blindly, in a manner not unlike an API and perhaps even more so. Second, this method also involved creating services and drivers that would require learning more about the in-depth workings of Windows, something that, again, was beyond the scope of this project.

As I began to research the NetDefender hooking method, my first task was to insure that it was in working order. NetDefender properly blocked traffic by filtering based on an IP address and protocol. However, it failed to filter based on ports: no matter what port it was given, the firewall automatically blocked all traffic from the related IP address. Deciding that this was not a satisfactory performance, although it worked far better than Firewall PAPI, I began researching to find another method for my firewall implementation. This led me to the Windows Filtering Platform, which will be referred to as "WFP" for the remainder of this paper.

## 3.3 Windows 7 and Windows Filtering Platform

There were many changes during the summer of 2010 since my initial research a year before. I personally obtained a computer with Windows 7, and Carson-Newman upgraded to Windows 7 as well, providing me with ample resources to

design a firewall for the Windows 7 platform. Also, Windows 7 proved to be a success since its release, rapidly gaining a foothold in the market share and continuously growing since then [11] [13] [14]. As such, WFP became not only a method that I could access but also one that could be used by a large and growing proportion of computer users. WFP was also an attractive option because of the large amount of support and documentation available.

WFP was initially introduced with Windows Vista as a new way to interact with the networking facets of the computer at a deeper level than previous version of Windows, including the API methods provided in Windows XP. WFP promotes a more open approach to network security, allowing the user to easily access the filtering engine instead of obscuring it. After its initial release, while some security companies were hesitant to switch their entire method of deploying a firewall, many other companies were more than willing to put their full support behind it, including McAfee who proclaimed that all of their future products would use WFP [12]. Those that were hesitant felt that by making the inner workings of network security open to the average user, hackers and other criminals would have a better chance at penetrating their defenses [12]. Overall however, reviews have been generally positive for WFP, both for its security and ease of use.

### 3.3.1 Kernel and User Modes

Code is usually operated in one of two different modes, either kernel or user mode, as a security and protection measure in order to protect important services [15]. There are actually four modes available, though the other two are rarely ever used. When the computer executes a task on behalf of an user application, the system

is said to be in "user mode". However, if an user application wishes to access a service that is part of the operating system, it temporarily switches to "kernel mode" to execute the function.

WFP can operate in either of these modes. While kernel mode provides more depth and control than the user mode does, it also requires the user to have an extensive knowledge of computer architecture and the Windows operating system. Like NetDefender's hooks, kernel mode in WFP requires the writing of drivers named "callout drivers" and, for the same reason as NetDefender, the decision to interact with the engine in the user mode API was made based on the amount of time, spanning months, it would take to research a relatively unrelated topic [4]. The user mode still provides the amount of control needed for this project and even excels in many places compared to the Firewall API, such as using the same engine for both packet and application filtering as opposed to using two separate APIs as originally planned.

### 3.3.2 Packet_Filter Examination

A sample open-source program for using WFP to filter packets based on IP address, simply named "`Packet_Filter`", was used to get a general understanding of the structure and order of operation of WFP. The Microsoft Developers Network, commonly referred to as MSDN, provides descriptions for many functions as well as sample code, but there was no structured piece of code that showed how each function interacted and built on top of one another. `Packet_Filter` not only did just that but it did so with no extra code for unnecessary functions, not even a GUI as it implements a command line interface (CLI). Though it was originally

built for Windows Vista, the changes Windows 7 made to WFP only enhanced its abilities and functions: they neither took functions away nor changed them substantially beyond bug fixes. As such, it promised to be highly useful for the beginning stages of this project.

The first step was a quick check to insure that the program did indeed block properly based on IP addresses. Though there were a few errors and bad programming choices, the functions executed successfully. They were also quite clear and easily deciphered. First, the program must open a session with the filtering engine by calling `FwpmEngineOpen0`, which returns a handle to the engine. The handle is some obscure reference to the engine: the developers purposely do not inform the user how the handle works as a form of encapsulation, usually for security reasons. It is the program's link to the engine, creating an interface. Next the interface must be bound to a sublayer that is attached to the engine using `FwpmSubLayerAdd0`. The sublayer is then given a GUID (**G**lobally **U**nique **Id**entifier) for any future operations, providing the program a way to identify the sublayer from any others that may already be implemented in the engine. Finally, the filter conditions are populated and added to the sublayer via the `FwpmFilterAdd0` command. Once this is done the firewall is active and begins to filter out packets based on the conditions provided.

### 3.3.3 Packet Filter Implementation

The packet filter portion of my firewall was thus implemented using the source code as a guideline. Much like Firewall PAPI, I had to make sure certain options, such as which Common Language Runtime support to use, were set properly.

Unlike Firewall PAPI, however, I was also required to download the Windows 7 Software Development Kit (SDK). The WFP functions were not included with my version of Visual Studio, and I had to manually link them to my project in order to use any of them. These functions, stored in library and DLL files, are necessary for the program to compile and run properly.

My packet filter was written to function in a similar manner to `Packet_Filter`, however there were numerous code corrections and safeguards added. The main changes were to allow filtering based on the port number and to block on both the inbound and outbound layer, as opposed to only the inbound. One of the layers I used was the `FWPM_LAYER_OUTBOUND_TRANSPORT_V4` layer for outgoing traffic. The other was the `FWPM_LAYER_INBOUND_TRANSPORT_V4` layer for incoming traffic. These layers can filter the packets before they are processed by any program. As such, this insures that the packets are blocked before they are allowed to bind to anything or be used in any way.

Safeguards included insuring that the conditions were filled in the proper order and preventing the user from attempting to start the firewall while it was already running, along with the implementing of transactions. A transaction allows a program to execute multiple steps without another process interrupting it, in other words atomically, to prevent errors. In the case of the packet filter, I used transactions due to each condition being added to two filters and the possibility of the data being mismatched depending on order of execution. Though this would rarely ever occur, with security programs one must insure for all possibilities. In order to do this, the transaction is committed once both filters are filled. I also added a basic GUI to increase functionality and make testing easier. Initial

versions only had buttons to start and stop the firewall with text boxes to indicate if the firewall was running, stopped, or an error occurred. The final version of the GUI is shown in Figure 3.1.


**Figure 3.1: The final GUI of the firewall**

The test runs were a complete success. I was able to block traffic based on an IP address, port, and protocol. For example, I was able to successfully block access to a website via a web browser if I used the TCP protocol, but I was still able to use the ping command since it uses the ICMP protocol. I could also block web access via blocking port 80, yet could still use other functions that involved other ports over the TCP protocol. Once the firewall was stopped, connections were able to be reestablished and the packets flowed to their destinations normally.

The packet filtering section of my firewall worked perfectly.

### 3.3.4 Application Filter Implementation

My next step in developing my firewall was to block by application rather than by IP address. This is a very useful ability, as it prevents any unsafe or undesired applications from making any connection on the network. I did not feel the need for any sample programs for the application filter as the majority of the commands are exactly the same as the packet filter. Through research on MSDN, I learned that every application is assigned an Application ID by Windows and that I could retrieve them using the `FwpmGetAppIdFromFileName0` command. WFP uses these IDs to identify what applications are accessing the network and are bound to which port. `FwpmGetAppIdFromFileName0` uses the path to the application's executable file to retrieve the ID and place it in a data structure. This structure is then used to check data as it comes to the `FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_VX` layer, where $X$ can be either 4 or 6 for each respective IP version. This layer checks programs before they bind to a port and, if the ID matches a condition, prevents the program from doing so. Again, I implemented a transaction to insure no errors would occur while changing the filter information.

My initial tests were again a success. Using a hard coded application path for Internet Explorer, I was able to keep it from accessing any network resources, yet it could still work on local resources such as a saved web page. No matter what URL was entered, Internet Explorer always immediately returned an error that it could not connect to the server. After this I began coding a method to search through the entire hard drive for all applications. For simplification reasons I chose

to search only for .exe executables, though there are numerous other executable extensions. To do this I employed a recursive searching algorithm, one that loops on itself to complete the process, using the `Directory` structure. Through it I was able to obtain the names and attributes of files and folders. After examining all files in a folder for the desired extension, the function then writes the filepath of any matches to a text file using the `StreamWriter` class for later use.

Initial tests of this method were fraught with complications. While searching, my program was denied access to many locations on my hard drive. However this was easily rectified by changing my user permissions to give me full access to these folders, as my program was running with the same access levels that my user account was. I then noticed that some folders were getting bypassed: this was a result of having system folders hidden, as the search passed over hidden folders, which was also easily fixed. However, a major bug was discovered after doing this; my program would constantly give me error messages that the filepath was too long. Knowing that this should not be the case, I debugged my program in order to find out which files were giving it the error. I quickly found that the search function was constantly looping over what Microsoft calls "junction points", also known as "symbolic links". Junction points are essentially folder shortcuts and are used in Windows 7 to aid in backwards compatibility for programs designed for earlier versions of Windows while insuring all related data, such as the application data, is stored in the same location for easy access and organization.

For example, in Figure 3.2 there are two folders: a normal folder named "Blue" and a junction point named "Red". The folder Red is also hidden, as most system junction points are. If folder Blue is opened, as in Figure 3.3, there is

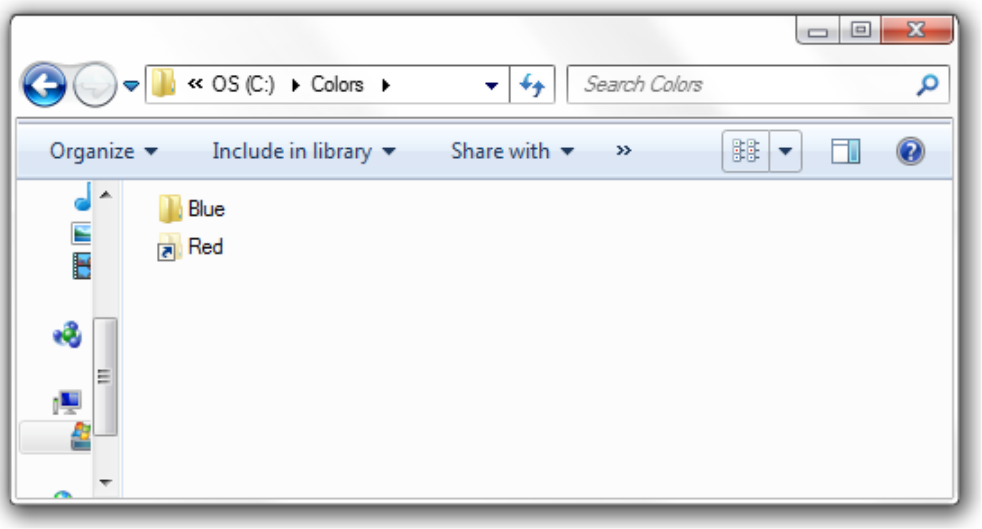


**Figure 3.2: The Colors folder, containing the folder Blue and hidden junction point Red**

a text file named "Blue1" stored there. The filepath for this text file would be "`C:\Colors\Blue\Blue1.txt`". However, if the junction point Red is used, as seen in Figure 3.4, one can see that the exact same text file Blue1 available. Yet the filepath for the text file this time would be "`C:\Colors\Red\Blue1.txt`". Both of these filepaths are valid and lead to the same file. This problem caused by junction points is twofold. First, time is wasted searching junction point filepaths that lead to the same files as the "real" filepath does, wasting time and processor power on fruitless searches. The second problem is demonstrated below. Suppose a junction point named "Color2" is added to the Blue folder and links back to the Colors folder, as is done in Figure 3.5. When the Color2 path is followed, the path becomes "`C:\Colors\Blue\Color2\`", which yields the same content as "`C:\Colors\`". What has happened is a loop in the file directory structure has been created. This path can be followed for an essentially infinite amount of

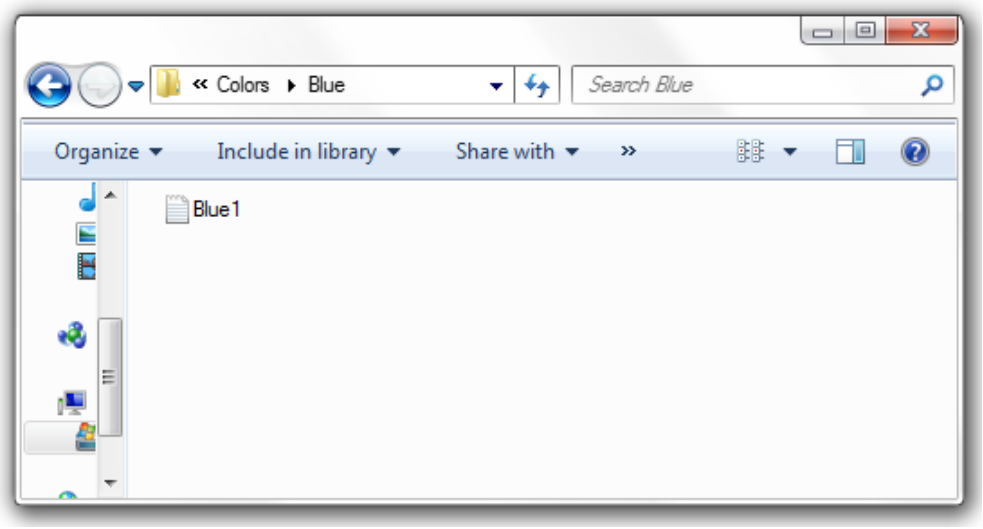


**Figure 3.3: The Blue folder, containing the text file Blue1**

steps: that is, until the size limit for the filepath is finally exceeded, as described previously. The system folders within Windows 7 actually contain these built-in loops, which was causing the errors I encountered. The easiest and most efficient method to avoid these problems is to completely ignore junction points, as all junction points do is provide a link to something that can be accessed from a different path: no new content is available by following the so-called symbolic links. My research led me to the `GetFileAttributes` function, which is able to get the properties, including whether it is a junction point, from files and folders. Thus any folder that is a junction point is ignored by the firewall as it gathers the executable files.

After implementing this function only one other error occurred: the System Volume Information folder still gave me access denied messages because I did not have any permissions for it. However, no matter what method I tried, I could not

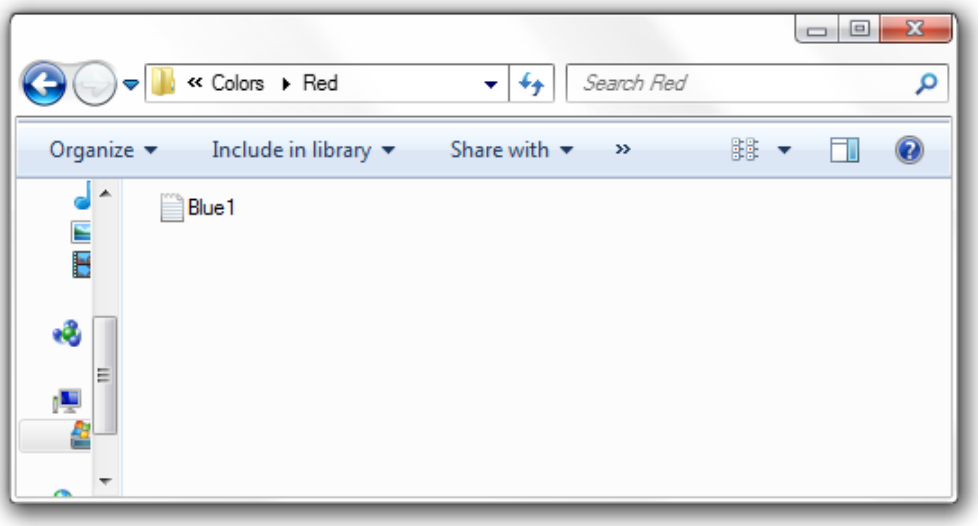


**Figure 3.4: The Red junction point, containing the text file Blue1**

gain access to this folder, not even to see its contents using Windows Explorer and attempting to change permissions manually. As such, I was forced to ignore this folder in my search algorithm. Given that it is a system folder that stores system restore information and cannot be accessed by anything but the operating system, the risk of a malicious program being stored there is almost nonexistent.

Once these errors were cleared up, my search function correctly found and listed the paths to all applications on my computer. I then made additions to my user interface to generate the list of application paths as the search is ongoing, storing them in a list box for the user to interact with. The user need only to check the application they wish to block and start the firewall to prevent the selected application from accessing network resources. I also added the ability to read from the text file I stored the filepaths in, using the `StreamReader` class, to save time on future tests and to be more user friendly. The user can choose to

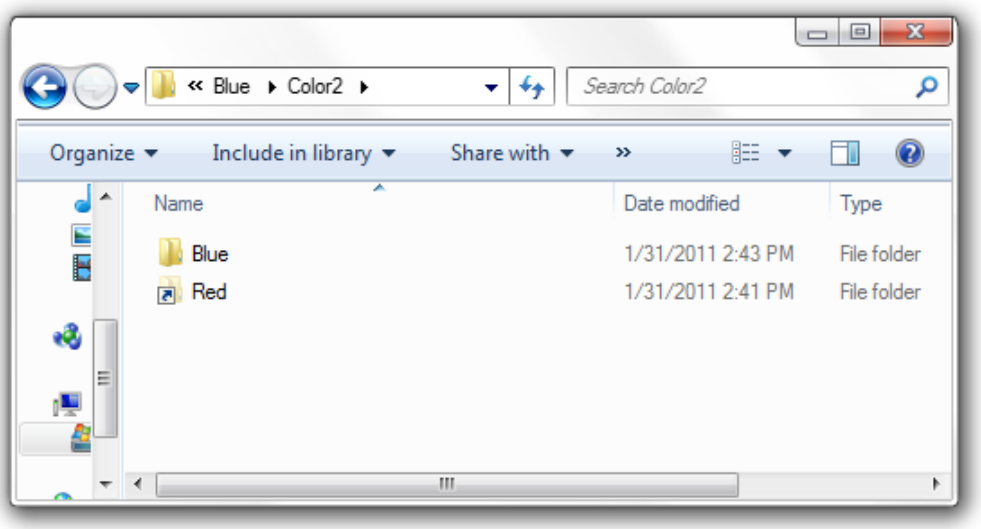


**Figure 3.5: The newly created Color2 junction point**

start a fresh search, which erases the current application list, or to load the list directly from the file to save time and resources.

Passing the filepaths from the checklist to the firewall turned out to be a complicated process. The `Directory` structure returns the filepaths as a string of type `System::String`, which is the same used in the forms controls for the user interface. However, WFP requires the application paths to be in `PCWSTR` format, which is a pointer to a string of 16 bit Unicode characters [10]. To make this conversion, the filepaths had to first be converted to a character pointer type using the `StringToHGlobalAnsi` command, which was also used to format them for the `GetFileAttributes` function. Once this was done, the filepath was then formated to a wide character format, which is a subtype of `PCWSTR`, by the `mbstowcs` function. The filepath is then properly formated and accepted by the firewall.

My second round of tests on these functions was also a success. I was able to pick any application from my checklist and block its network access. For example, I was able to block Internet Explorer from accessing network resources, yet it was still able to access locally stored web pages. Other programs, such as Mozilla Firefox, were still able to access network resources, using the same protocol, IP address, and port information that Internet Explorer was attempting to use. However, one noticeable security loophole was spotted during this testing phase. If a program was already bound to a port when the firewall was started and was constantly listening on it, the firewall had no effect since it only checked and blocked applications when the port was initially bound. While it is rare that applications do this due to ports often being used by multiple programs almost simultaneously, some such as Windows Live Messenger do. If I attempted to block its access while it was connected, the firewall had no effect and the program continued properly. However, if it was activated while it was not connected or if the program was disconnected while the firewall was running, Windows Live Messenger would be properly blocked. Unfortunately, no other WFP layers that accept Application IDs as conditions would affect this either. In order to block these applications, more code would be required. This code was beyond the scope of this project, as it involved delving into other in-depth Windows concepts, taking far too much time [3].

### 3.3.5 Port Scanning Defense Implementation and Difficulties

The previous two methods, filtering by IP address and Application ID, are active defensive measures. They only benefit security if one knows what programs to block or what IP is attacking the computer. For a firewall to be truly effective, it needs to be able to block basic intrusions, namely port scanning, from unknown sources. This type of defense is a passive defense, as the firewall is not actively defending the computer from a specific target. By using passive defenses, the computer is less likely to be attacked by a random hacker and has a better defense against more advanced hacking tools. Most hackers are only interested in either a lucrative or easy prize: by appearing to not be worth their time, they will pass over the computer in hopes of finding a weaker one.

The main types of scans that I wished to stop are known as stealth scans. In stealth scans, the attacking computer attempts to connect to the target computer in a normal manner to each port in sequence. If the target computer responds with an acknowledgment, the attacker knows that the selected port is open and can be used to attack. The attacking computer will then respond with a packet using the reset flag to prevent a connection from being fully established and thus logged. If the target computer sends back a packet with the reset flag set, then the attacker knows that the port may be closed or filtered. No matter what the response the attacking computer gets, with this method the transactions will not be logged by the target computer [16].

This is a security problem that exists in the TCP protocol, though similar weaknesses exist in the UDP protocol as well. In order to counteract this, the

firewall is designed to not send out any reset or destination unreachable responses. Because of this, the attacker will not know which ports are open from a stealth scan. This forces the hacker to either move on to a new target or to try more active types of scans against the target. If he or she decides on the latter then the connection, including the IP address information of the attacker, will be logged and active measures can be taken against them. As such, a passive defense against stealth scans is an extremely important feature.

To implement this, I used the MSDN library as a background guide. I implemented it in the same method as I did in the packet filter section, including the use of a transaction. Since four filters are used, one set each of IPv4 and IPv6 for UDP and TCP, the chance of an error is significantly higher than it was with the packet and application filters. For the TCP sets I used a built-in callout driver named `FWPM_CALLOUT_WFP_TRANSPORT_LAYER_VX_SILENT_DROP`, where $X$ is either 4 or 6 for each protocol version, to enable the dropping of packets. For UDP scanning prevention the `FWPM_LAYER_OUTBOUND_ICMP_ERROR_VX` layer is used, again with $X$ being either 4 or 6.

However, when I went to test this feature, the filters failed to block any port scanning when activated. I then ran a test from the command prompt using a Windows Filtering Platform Diagnostic tool to log WFP changes and transactions. The log file generated showed that for some unknown reason the filters were not being added to the engine properly, although the sublayer was. This contradicted the error messages generated by the functions adding the filters, as they were all "success" error codes. I attempted to consult MSDN for a solution, however the database revealed nothing and the specialist on the forums, after pointing

me to the diagnostic tool, failed to follow up on my error. I therefore had to discontinue my work on this feature, though passive defense measures are one of the core features of firewalls. I left my code and program with the non-functioning features still available for viewing and testing.

## 3.4 Security Testing

After all the methods of my firewall were implemented and running correctly, I then tested my firewall's defense against potential hackers. To do this, I connected my computer to another Windows 7 computer using a crossover cable, which allows for direct connections without the use of a hub or switch through the Ethernet port. Both computers were disconnected from any other networks to insure that there was no possibility of any of my actions affecting an outside target. The default Windows Firewall and any other security software were also disabled during the tests.

### 3.4.1 Blocking Applications

My first test was to insure that I could block application installed on my computer from accessing the network. The firewall successfully blocked all attempts from the designated application, be it harmless or malicious, to bind to a port and communicate on the network. Though the firewall works as intended, it is worth noting that there are two major problems with this method.

The first is that, as previously noted, applications that are currently bound on a port cannot be blocked until they unbind and attempt to bind again. Yet this is

not a major concern, as in such a case a program could just as easily be forcefully terminated, allowing the firewall's block to take effect. The other problem is that many types of malicious software often use other programs to access network resources. For example, a Trojan may be installed on a system and blocked from accessing the network by the firewall. However, the Trojan may instead use a different program, such as Internet Explorer, to connect to a specific IP address and bypass the block, enabling the Trojan to essentially have full network access. The only method of prevention for this would be to prevent applications from opening any other file or application. This is a dangerous path, however, as many applications, including system ones, depend on one another to complete tasks. Regardless, such a defensive measure lies more in the realm of file protection and security. This type of defense is often found in services such as anti-virus scanners and is thus outside the scope of this project. The end result is that the firewall blocks applications just as designed and in an effective measure, though it can be bypassed.

### 3.4.2 Port Scanning

My first set of tests for my packet filterer were done using Nmap, the port scanning tool, to try to gather information when the firewall was off and when the active defenses were online. Starting with a quick stealth scan of the 100 most common ports while the firewall was off, Nmap was able to identify 8 open ones. The most noticeable of these was port 139, which is the NetBIOS service. This service has been used constantly to attack Windows-based computers for years. No other information was able to be retrieved by this type of scan: the other 92 ports were

listed as filtered, and thus could be anything. When the same scan was run with active defenses blocking the IP of the scanning computer, the results were much improved: all scanned ports were declared as filtered and no further information could be found. As such, an attacker would have gained no information about the computer.

I then ran a more intrusive scan without blocking the attacking computer. This time Nmap scanned 1,000 common ports and found 9 open ports, labeling the rest as filtered. Nmap then executed a trace-route to map the path to the target computer. Given the two computers were connected directly by a cable, the route it used was rather obvious, but it executed successfully nonetheless. It was also able to fingerprint my computer as either Windows Vista, Windows Server 2008, or Windows 7 with 100% accuracy. With IP blocking on, however, Nmap was foiled at every turn. All 1,000 ports were listed as filtered and no route could be established, as no packets were able to reach their destination. Nmap was also unable to fingerprint my computer, labeling it as simply an unknown operating system.

### 3.4.3 Enumeration

I then began using NetScanTools Pro 2010 to attempt to enumerate my computer, using the same style of tests: one while the firewall was off-line and one while it was actively defending itself. Part of the enumeration process includes port scanning, and in both tests the results were the same. The reason behind this is that NetScan contains a packet crafter, allowing the user to manually encode the packets it sends out, including the IP address information. Since NetScan has

this ability, it is likely used automatically in the port scanning process to change the IP of the packets it sends out. This would effectively bypass any active IP blocking defense, as is used by my firewall. The only method to prevent this type of scan from succeeding is a passive defense, as described in section 3.3.5.

Other tests met with similar fates: either NetScanTools was able to extract the information it wanted, such as in the operating system fingerprinting, or it was stopped by built-in security from the services it was trying to exploit, such as NetBIOS. My firewall was unable to stop the attacks with its active defensive measures thanks to the packet crafting ability provided by NetScan. I have used active defenses such as IP blocking to prevent specific users from connecting to a server for several years. These defenses were always relatively easily bypassed through dynamic IP addresses; simply reseting a router or reconnecting to one's Internet Service Provider (ISP) is all that is needed to fool these measures. I myself was often forced to monitor the server for hours, constantly manually blocking suspicious and similar IP addresses to prevent a malicious user from connecting to the server. The only way to properly block connections actively is if there was a unique identifier. In times past, a network card's Media Access Control (MAC) address, usually hard-coded into the network card, was often used as this identifier, but now even this can be spoofed and changed. This merely stresses the importance for firewalls to have passive defensive measures, such as port scanning prevention. In the world of packet crafting and dynamic IP addresses, active defenses are very limited and in most cases useful merely as a temporary defense.

## 3.5 Conclusion

Though many difficulties were encountered, most were eventually overcome and a working firewall was produced using Windows 7 and WFP. While the security tests did not produce the best results, they were expected given the lack of passive defenses and the style of attacks used. Overall, the firewall exceeded initial expectations but did not meet later ones. Even so, this firewall is still a useful tool and can be used to give an user easier and more control in securing their Windows 7 system.

# Chapter 4

# Reflections

## 4.1 Gaining a New Perspective on Programming

Through this project I learned many things, foremost being that no matter what kind of depth one attempts to delve into in creation of any program, especially one that relies heavily on the operating system such as a firewall, it will always be reliant on some degree to another function over which the programmer has no control. The only real change that can be made is to rely less on the functions as I did with switching from the Packet Filter API to WFP: both rely on an API to implement the rules that the user decides, but WFP allows more access and thus less blind reliance on some unknown system function.

I also learned a great deal about programming in general; being my first large scale program, this project showed me the necessity of many aspects that I often despised while programming simpler applications, such as naming practices and commenting code. These features definitely make it far easier for another pro-

grammer to read and understand one's code, even if he or she understands the language and all the commands. The simplicity of Object Oriented Programming also showed for this type of project, allowing me to concentrate on smaller functions and objects first, then gathering them together afterwards. Once the initial structure was in place, implementing new features and designs was a relatively simple process.

This was also the first time I programmed something that was operating system specific. Through this project I discovered the difficulties in relying on the operating system functionality. My experience with Windows XP was highly frustrating with the lack of documentation and the need to have a specific version in order for certain commands to work: something that is nigh impossible with the frequent updates that are released. Also, due to the use of Windows, many of the functions used had to be implemented after seeing multiple examples, as the code is not open-source so I could not examine it for myself. This created a lot of difficulty, resulting in a Catch-22 situation: functions could not be easily used without examples that could not be easily created without examples. While it is possible to figure out the usage of the functions without any sample code, MSDN can be very vague in its descriptions of what a function does and what output is generated.

## 4.2 Networking and Security

I learned much about the processes involved in both defending and attacking a computer. It is almost a paradox in the sense that both are easy and difficult

to do at the same time: once one overcomes the complexity involved in knowing what one is doing, the actual act of doing it becomes rather simplified. While this can be said of almost any topic, the gap between the two in networking is enormous: port scanning and packet crafting seem simple to one with knowledge of networking, while to others it is quite frankly nothing but gibberish, even to those in the field of Computer Science.

Another important concept learned through this project is the need to have multiple types of security to help defend against different attacks. Firewalls are well suited for the first line of defense, but if the firewall is broken it then becomes useless. This is where other security software, such as anti-virus and file protection applications come into play. Every layer of security may be independent from others, but each layer also covers the gaps the others leave behind. This is why the majority of software vendors offer their software as suite packages: one style of defense is not enough to fully protect any computer, much less high profile targets. While no computer connected to a network is ever completely safe from these attacks, the best method to secure a computer is by employing several different types of security programs to try to cover as many different methods of attacks as possible.

Overall, one should recognize the difficulty involved in making a working program of any type, especially one related to security. Programs dedicated to security are an even more likely target for abuse and other attacks, and as such they must be programmed to not only withstand these attacks on themselves but the systems they protect as well. If a normal program is broke, often the worst-case scenario is just that: the program does not work and everything else functions

normally. However with security programs, if the program is taken down in any way, the entire computer's or even the entire network's security can be at risk.

## 4.3 Final Words

Security in general is an important part of our society. So much important data is digitally stored that to not secure it would invite disaster. Credit card numbers, Social Security Numbers, financial records, medical records, and other important data are all stored on computers and need protection. Almost every major business has someone employed to deal with security threats. This is usually the head of the I.T. department, but it may also be a different worker. Even the government is not immune. Since a government is a bigger target with more valuable data, security is even more important for it. Common criminals, terrorists, or other nations may attempt to gain access to confidential information.

Firewalls are a large part of security and are often the first line of defense. As such, an in depth knowledge of firewalls will also give a greater understanding of security methods as a whole. With security being such a large part of the digital world today, it is essential for anyone entering the field of computers to have an understanding of security. Whether the person is a programmer or an I.T. staff member, security will be a vital part of one's job that can result in the success or failure of any company. Regardless of one's specialization in the Computer Science field, security will always be a factor in decision making.

Information security is the battlefield of the digital age. It is no mere analogy: there is a constant war over data and any military attack on another country

will likely be preceded by a digital one. Much like armor will always be needed to defend against the latest munitions, security tools must always be available to combat the latest hacking and cracking methods. Unfortunately, just like physical warfare, no amount of security can completely protect any computer from every attack. Yet by constant vigilance and with continuous improvement in one's defenses, one can deflect weaker attacks while softening the impact of greater ones. After all, the defenders need to be victorious every time to be truly successful, while the attackers need only win once.

# Bibliography


[1] David R. Mirza Ahmad. *Hack Proofing Your Network*. Syngress, 2002.

[2] Code Guru Forums. Debug assertion failed. Electronic, 2010.

[3] MSDN Forums. Blocking application that has already bound to port? Electronic, 2010.

[4] MSDN Forums. Windows driver programming and wfp. Electronic, 2010.

[5] Katie Hafner. *Where Wizards Stay Up Late: The Origins of the Internet*. Simon & Schuster, 1996.

[6] Gilbert Held. *Cisco Security Architectures*. McGraw-Hill, 1999.

[7] David Khan. *Seizing the enigma: the race to break the German U-boat codes, 1939-1943*. Barnes & Noble, 2001.

[8] Tennessee State Legislature. Legal resources. Electronic, 2010.

[9] Microsoft. Release notes for windows xp service pack 3. Electronic, 2010.

[10] Microsoft. Windows data types (windows). Electronic, 2010.

[11] Stat Owl. Windows, mac and linux usage / market share. Electronic, 2010.

[12] Neil J. Rubenking. Security in windows 7: Windows filtering platform. 2009.

[13] W3 Schools. Os statistics. Electronic, 2010.

[14] Net Market Share. Market share for browsers, operating systems and search engines. Electronic, 2010.

[15] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley and Sons, eight edition, 2009.

[16] Michael T. Simpson. *Hands-on Ethical Hacking and Network Defense*. Thomson, 2006.

[17] Keith Strassberg, Richard Gondek, and Gary Rollie. *Firewalls: The Complete Reference*. McGraw-Hill/Osborne, 2002.

[18] Michael E. Whitman, Herbert J. Mattord, and Richard D. Austin. *Guide to Firewalls and Network Security: with Intrusion Detection and VPNs*. Course Technology, 2009.

[19] Business Wire. Mcafee, inc. report reveals cyber coldwar, with critical infrastructure under constant cyberattack causing widespread damage. Electronic, 2010.

[20] Anne L. Young. *Mathematical Ciphers: from Caesar to RSA*. AMS Bookstore, 2006.

# Appendix A

# Appfilter.h Code

```
/*********************************************
Created by Terry Rogers on September 19, 2010

Application Filter Class
This file declares the functions are used in the
Application Filter class
*********************************************/



/********Includes********/
#include "stdafx.h"
#include <Winsock2.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <strsafe.h>
#include <fwpmu.h>
#include <list>

/********Definitions********/

//Error Codes
#define APP_ALREADY_RUNNING 11 //App Filter is already running
#define APP_NOW_RUNNING 1 //App Filter is now started
#define APP_ALREADY_STOPPED 10 //App filter is already stopped
#define APP_NOW_STOPPED 0 //App filter is now stopped
```

```
#define APP_FAILED -1 //Could not start or start the App filter

//Sublayer Name
#define APP_SUBLAYER_NAMEW L"APPDinsFirewall"

/********Structures********/

//Structure to store IP and ID information for the filter
typedef struct _aFILTERINFO {
    UINT64 FilterID4; //Filter ID for IPv4
UINT64 FilterID6; //Filter ID for IPv6
DWORD appID; //Application ID
wchar_t *appPath; //Application filepath
} aFILTERINFO, *PaFILTERINFO;

//List to hold the Filter Infos
typedef std::list<aFILTERINFO> aFILTERINFOLIST;


/********Class Declaration********/

class AppFilter
{

/********Private Declarations********/
//These functions and variables should only
//be used internally by the class
private:
    //Handle for the Filtering Engine
    HANDLE appEngineHandle;

    //GUID to identify the sublayer
    GUID appsubGUID;

//List containing filters
aFILTERINFOLIST aFilterList;

    /************************************************
Inputs: None
Returns: Error code indicating any errors
```

```
Post: The Filter Interface is created
*************************************************/
    DWORD appCreateInterface();

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is deleted
*************************************************/
    DWORD appDeleteInterface();

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is bound
*************************************************/
    DWORD appBindInterface();

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is unbound
*************************************************/
DWORD appUnbindInterface();

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is added to our interface
*************************************************/
DWORD appAddFilter();

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is removed from our interface
*************************************************/
DWORD appDelFilter();

public:
```

```
/********Public Declarations********/
//The functions and variables that can
//be accessed from outside of the class

//Boolean to indicate if firewall is
//currently running (true) or not (false)
bool apprunning;

/*************************************************
Inputs: None
Returns: None
Post: Our class is constructed, memory allocated
*************************************************/
    AppFilter();

    /*************************************************
Inputs: None
Returns: None
Post: Our class has been destructed
*************************************************/
    ~AppFilter();

    /*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Application filter is running and active
*************************************************/
    int appStartFirewall();

    /*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Application filter is stopped and inactive
*************************************************/
    int appStopFirewall();

/*************************************************
Inputs: Character pointer formatted Application path
Integer with application path length
Returns: None
```

```
Post: The app is added to our list of applications to block
**********************************************/
    void AddAppToBlockList(char* sPath, unsigned int length);
};
```

# Appendix B

# Appfilter.cpp Code

```
/*********************************************
Created by Terry Rogers on September 19, 2010

Application Filter Class
This file implements the functions we use in
the Application Filter class

*********************************************/


/********Includes********/
#include "Appfilter.h"


/********Functions********/

/*************************************************
Inputs: None
Returns: None
Post: Our class is constructed, memory allocated
*************************************************/
AppFilter::AppFilter(){
appEngineHandle = NULL; //Initialize the handle
//Allocate memory for GUID, fill with 0s
ZeroMemory(&appsubGUID, sizeof(GUID));
}
```

```
/*************************************************
Inputs: None
Returns: None
Post: Our class has been destructed
*************************************************/
AppFilter::~AppFilter(){
appStopFirewall();
}

/*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The application filter is running and active
*************************************************/
int AppFilter::appStartFirewall(){
//If the Application filter is already running...
if (apprunning) {
return APP_ALREADY_RUNNING; //Indicate filter is currently running
}
else{ //Filter is not running...
        //Create the application filter interface
if(appCreateInterface() == ERROR_SUCCESS){
            //Bind the application filter interface
if(appBindInterface() == ERROR_SUCCESS){
                //Add the filters
if (appAddFilter() == ERROR_SUCCESS) {
//Application filter is now running
            apprunning = true;

//Message indicating filter is running
return APP_NOW_RUNNING;
}
}
        }
    }
//Starting application filter failed at some point
return APP_FAILED;
}

/*************************************************
```

```
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The application filter is stopped and inactive
************************************************/
int AppFilter::appStopFirewall(){
//If the application filter is not running...
if (!apprunning) {
//Indicate filter is already stopped
return APP_ALREADY_STOPPED;
}
else {//Filter is running
if (appDelFilter() == ERROR_SUCCESS) {
aFilterList.clear(); //Clear our filter list
if(appUnbindInterface() == ERROR_SUCCESS) {
if(appDeleteInterface() == ERROR_SUCCESS) {
        //Firewall is no longer running
apprunning = false;

//Message indicated application filter is no longer running
return APP_NOW_STOPPED;
            }
    }
}
    }
//Stopping application filter failed at some point
return APP_FAILED;
}

/************************************************
Inputs: String formatted Application path
Returns: None
Post: The app is added to our list of applications to block
************************************************/
void AppFilter::AddAppToBlockList(char* cPath,
  unsigned int length){

//Structure to store application info
aFILTERINFO Appfilter = {0};

//Initialize wide character format variable for filepath
```

```
wchar_t *wPath = (wchar_t *)malloc(sizeof(wchar_t) * length);

//Convert the filepath into wide character format
int result = mbstowcs(wPath, cPath, length);

if (result != -1){ //If conversion was successful

//Store the newly formatted path
Appfilter.appPath = wPath;

//Add the structure to the list
aFilterList.push_back(Appfilter);
}
}

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is created
*************************************************/
DWORD AppFilter::appCreateInterface(){

//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

    //Create packet filter interface by
//opening a session with the filtering engine
    ErrorCode =  FwpmEngineOpen0(//Must be NULL
 NULL,
 //Specifies authentication service to use
 RPC_C_AUTHN_WINNT,
 //Authorize credentials for filter engine (optional)
 NULL,
 //Session specific parameters (optional)
                                 NULL,
 //Engine handle set
                                 &appEngineHandle );

//Error code indicating any errors or success
return ErrorCode;
```

```
}

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is deleted
*************************************************/
DWORD AppFilter::appDeleteInterface(){

//ErrorCode for WFP functions, default is bad command
DWORD ErrorCode = ERROR_BAD_COMMAND;
//If our engine handle is still set properly (not NULL)
if(appEngineHandle != NULL){
//Close the interface
        ErrorCode = FwpmEngineClose0(appEngineHandle);
//Reset the Engine Handle (to NULL)
        appEngineHandle = NULL;
    }
//Error code indicating any errors or success
    return ErrorCode;
}

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is bound
*************************************************/
DWORD AppFilter::appBindInterface(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;
//Remote Procedure Call status
    RPC_STATUS rpcStatus = {0};
//Sublayer structure
    FWPM_SUBLAYER0 SubLayer = {0};

    //Create a GUID for the SubLayer for easy access
    rpcStatus = UuidCreate(&SubLayer.subLayerKey);
    if (rpcStatus == NO_ERROR){
/*Save the GUID for future use. Copies memory directly from
SubLayer to appsubGUID from beginning of subLayerKey
```

```
to the size of subLayerKey*/
        CopyMemory(&appsubGUID,
                   &SubLayer.subLayerKey,
                   sizeof(SubLayer.subLayerKey));

        //Fill in the SubLayer information
        SubLayer.displayData.name = APP_SUBLAYER_NAMEW;
        SubLayer.displayData.description = APP_SUBLAYER_NAMEW;
        SubLayer.flags = 0;
        SubLayer.weight = 0x100; //Importance of the sublayer

        //Add the SubLayer to the interface
        ErrorCode = FwpmSubLayerAdd0(appEngineHandle,
                                     &SubLayer,
                                     NULL);
}
//Error code indicating any errors or success
return ErrorCode;
}

/***************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is unbound
***************************************************/
DWORD AppFilter::appUnbindInterface(){
//Delete the sublayer from the interface
    DWORD ErrorCode = FwpmSubLayerDeleteByKey0(appEngineHandle,
                                               &appsubGUID);
//Fill the GUID memory space with 0s
    ZeroMemory(&appsubGUID, sizeof(GUID));

//Error code indicating any errors or success
    return ErrorCode;
}

/***************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is added to our interface
```

```
************************************************/
DWORD AppFilter::appAddFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

//Make sure we have filters to add
    if(aFilterList.size() > 0){
//Create an iterator to keep track of what filter we are on
aFILTERINFOLIST::iterator numFilter;
//For each filter in our list
        for(numFilter = aFilterList.begin();
numFilter != aFilterList.end();
numFilter++){
//Constant number of conditions required
const UINT8 numCond = 1;
//Filter structure for filter info
FWPM_FILTER0 Filter;
//Fill filter structure with 0s
ZeroMemory(&Filter, sizeof(FWPM_FILTER));
//Filter condition structure for
//trigger condition info (1 for each condition)
            FWPM_FILTER_CONDITION0 Conditions[numCond];
//Fill filter condition structures with 0s
ZeroMemory(&Conditions,
   sizeof(FWPM_FILTER_CONDITION)*numCond);
//Byte blob structure for storing app data
FWP_BYTE_BLOB *appBlob = NULL;
//Boolean to check if the transaction is in progress
bool txninProgress = FALSE;

//Begin the transaction
ErrorCode = FwpmTransactionBegin0(appEngineHandle, 0);
txninProgress = TRUE;

//Get data from the filepath given, store in appBlob
DWORD result = FwpmGetAppIdFromFileName0(numFilter->appPath,
 &appBlob);

/*Fill condition in with needed information for Application Data*/
            //Look in the APP_ID field of the packet
```

```
            Conditions[0].fieldKey = FWPM_CONDITION_ALE_APP_ID;
//Trigger when APP_ID is equal to our filter
            Conditions[0].matchType = FWP_MATCH_EQUAL;
//We are comparing Byte Blobs that contain our app info
            Conditions[0].conditionValue.type = FWP_BYTE_BLOB_TYPE;
//The appBlob info, including Application ID
            Conditions[0].conditionValue.byteBlob = appBlob;

            /*Fill filter in with needed information*/
//GUID for easy access and deletion
            Filter.subLayerKey = appsubGUID;
//Name of Filter for displaying only
            Filter.displayData.name = APP_SUBLAYER_NAMEW;
//Inspect when resources (ports) are assigned (bound)
            Filter.layerKey =FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4;
//Block applications that meet our
//condition from binding to a port in V4
            Filter.action.type = FWP_ACTION_BLOCK;
//Let the engine auto assign the weight of the filter
            Filter.weight.type = FWP_EMPTY;
//Conditions to be fulfilled for action
//to be taken (previous condition struct)
            Filter.filterCondition = Conditions;
//Number of conditions on our filter
            Filter.numFilterConditions = numCond;

//Add our filter to the engine
            ErrorCode = FwpmFilterAdd0(appEngineHandle,
                                       &Filter,
                                       NULL,
                                       &(numFilter->FilterID4));

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;
```

```
//Change the filter layer to the V6 layer
            Filter.layerKey =FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6;

//Add our filter to the engine
ErrorCode = FwpmFilterAdd0(appEngineHandle,
                                    &Filter,
                                        NULL,
                                        &(numFilter->FilterID6));

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

/*If everything succeeded, then we add the filters atomically
**At this point we are actually adding the filters. Previous
**code is now executed*/
ErrorCode = FwpmTransactionCommit0(appEngineHandle);
txninProgress = FALSE;

//Cleanup operation if any filter additions failed
CLEANUP:
//if transaction is in progress and we had an error
//(from a goto) then abort the transaction (do nothing)
if ((txninProgress) && (ErrorCode != ERROR_SUCCESS)){
//Abort the transaction currently in progress
FwpmTransactionAbort0(appEngineHandle);
}
}
}
//Error code indicating any errors or success
return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
```

```
Post: The Filter is removed from our interface
**************************************************/
DWORD AppFilter::appDelFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

//Make sure our list is not empty
if(aFilterList.size() > 0){
//Create an iterator to go through our list
aFILTERINFOLIST::iterator numFilter;
//Delete all the filters we added
for(numFilter = aFilterList.begin();
numFilter != aFilterList.end();
numFilter++){
//Delete the filter via the FilterID4 (GUID for IPv4)
ErrorCode = FwpmFilterDeleteById0
(appEngineHandle,
                          numFilter->FilterID4);
//Delete the filter via the FilterID6 (GUID for IPv6)
ErrorCode = FwpmFilterDeleteById0
(appEngineHandle,
                          numFilter->FilterID6);
            //Reset the ID fields
numFilter->FilterID4 = 0;
numFilter->FilterID6 = 0;
//If an error is encoutered, stop trying
//to delete filters and report it
if (ErrorCode != ERROR_SUCCESS) break;
}
}
//Error code indicating any errors or success
    return ErrorCode;
}
```

# Appendix C

# Packetfilter.h Code

```
/*********************************************
Created by Terry Rogers on September 4, 2010

Packet Filter Class
This file declares the functions we use in the
Packet Filter class, including port scanning defense

*********************************************/

/********Includes********/

#include "stdafx.h"
#include <Winsock2.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <strsafe.h>
#include <fwpmu.h>
#include <list>


/********Definitions********/

//Error Codes

//Packet filter is already running
#define PF_ALREADY_RUNNING 11
```

```
//Packet filter is now running
#define PF_NOW_RUNNING 1
//Packet filter is already stopped
#define PF_ALREADY_STOPPED 10
//Packet filter is now stopped
#define PF_NOW_STOPPED 0
//Could not stop or start packet filter
#define PF_FAILED -1
//Port Scanning defense is already running
#define PS_ALREADY_RUNNING 22
//Port Scanning defense is now running
#define PS_NOW_RUNNING 2
//Port Scanning defense is already stopped
#define PS_ALREADY_STOPPED 20
//Port Scanning defense is now stopped
#define PS_NOW_STOPPED -20
//Could not stop or start port scanning defense
#define PS_FAILED -2

//Protocols

//Internet Control Message Protocol
#define ICMP 1
//Internet Group Management Protocol
#define IGMP 2
//Bluetooth Radio Frequency Communications Protocol
#define BLUET 3
//Transmission Control Protocol
#define TCP 6
//User Datagram Protocol
#define UDP 17
//Internet Control Message Protocol Version 6
#define ICMP_6 58
//Reliable Multicast protocol
#define RM 113

//Sublayer Names
#define PF_SUBLAYER_NAMEW L"PFDinsFirewall"
#define PS_SUBLAYER_NAMEW L"PSDinsFirewall"
```

```
//Default Subnet Mask
#define SUBNET_MASK 0xffffffff

/********Structures********/

//Structure to store IP and ID information for the filter
typedef struct _pFILTERINFO {
    BYTE ByteAddr[4]; //IP address separated by byte for debugging
    ULONG HexAddr; //IP address in Hexadecimal format
    UINT64 OutFilterID; //Outbound filter ID
UINT64 InFilterID; //Inbound filter ID
UINT16 Port; //Port indicated with IP address
UINT8 Protocol; //Protocol indicated
} pFILTERINFO, *PpFILTERINFO;

//List to hold the Filter Infos
typedef std::list<pFILTERINFO> pFILTERINFOLIST;


/********Class Declaration********/

class PacketFilter
{

/********Private Declarations********/
//The functions and variables should only
//be used internally by the class
private:
    //Handle for the Filtering Engine (packet filtering)
    HANDLE pfEngineHandle;

//Handle for the Filtering engine (port scanning)
HANDLE psEngineHandle;

//GUID to identify the sublayer for the packet filter
    GUID pfsubGUID;

//GUID to identify the sublayer for port scanning prevention
GUID pssubGUID;
```

```
    //List containing filters
    pFILTERINFOLIST pFilterList;

//Array for the port scanning filter ID keys
UINT64 psFilterID[4];


/*PORT SCANNING DEFENSE FUNCTIONS*/
    /*****************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is created
*****************************************************/
    DWORD psCreateInterface();

/*****************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is deleted
*****************************************************/
    DWORD psDeleteInterface();

    /*****************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is bound
*****************************************************/
    DWORD psBindInterface();

/*****************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is unbound
*****************************************************/
DWORD psUnbindInterface();

/*****************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is added to our interface
```

```
************************************************/
DWORD psAddFilter();

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is removed from our interface
************************************************/
DWORD psDelFilter();


/*PACKET FILTER FUNCTIONS*/
    /************************************************
Inputs: cIPAddr: Character pointer formatted IP address
nStrLen: Length of the IP address string
bIPAddr: Byte array for the IP address
hIPAddr: Unsigned long that contains the IP in hexadecimal format
Returns: True (IP formatted successfully) or
 False (Invalid IP or failure to format)
Post: The FilterInfo contains the IP address in
  byte array and hexadecimal format
************************************************/
    bool FormatIPAddr(char* cIPAddr,
  UINT nStrLen,
  BYTE* bIPAddr,
  ULONG&
  hIPAddr);

    /************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is created
************************************************/
    DWORD pfCreateInterface();

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is deleted
************************************************/
```

```
    DWORD pfDeleteInterface();

    /************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is bound
************************************************/
    DWORD pfBindInterface();

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter Interface is unbound
************************************************/
DWORD pfUnbindInterface();

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is added to our interface
************************************************/
DWORD pfAddFilter();

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is removed from our interface
************************************************/
DWORD pfDelFilter();

public:
/********Public Declarations********/
//The functions and variablescan be accessed from
//outside of the class

//Boolean to indicate if packet filter is
//currently running (true) or not (false)
bool pfrunning;

//Boolean to indicate if port scanning defense
```

```
//is currently running (true) or not (false)
bool psrunning;

/*************************************************
Inputs: None
Returns: None
Post: Our class is constructed, memory allocated
*************************************************/
    PacketFilter();

    /*************************************************
Inputs: None
Returns: None
Post: Our class has been destructed
*************************************************/
    ~PacketFilter();

    /*************************************************
Inputs: Character pointer formatted IP address,
Unsigned integer (8 bits) protocol
Unsigned integer (16 bits) port
Returns: None
Post: The IP is added to our list of IPs to block
*************************************************/
    void AddIPToBlockList(char* cIPAddr, UINT8 proto, UINT16 prt);

    /*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Packet Filter is running and active
*************************************************/
    int pfStartFirewall();

    /*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Packet Filter is stopped and inactive
*************************************************/
    int pfStopFirewall();
```

```
    /************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Port Scanning Defense is running and active
************************************************/
    int psStartFirewall();

    /************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The Port Scanning Defense is stopped and inactive
************************************************/
    int psStopFirewall();
};
```

# Appendix D

# Packetfilter.cpp Code

```
/*********************************************
Created by Terry Rogers on September 4, 2010

Packet Filter Class
This file implements the functions we use in the
Packet Filter class, including port scanning defense

*********************************************/

/********Includes********/
#include "Packetfilter.h"


/********Functions********/

/*************************************************
Inputs: None
Returns: None
Post: Our class is constructed, memory allocated
*************************************************/
PacketFilter::PacketFilter(){
//Initialize the handles
pfEngineHandle = NULL;
psEngineHandle = NULL;
//Allocate memory for GUID, fill with 0s
ZeroMemory(&pfsubGUID, sizeof(GUID));
ZeroMemory(&pssubGUID, sizeof(GUID));
```

```
}

/*************************************************
Inputs: None
Returns: None
Post: Our class has been destructed
*************************************************/
PacketFilter::~PacketFilter(){
//Stop the packet filter before closing the app
pfStopFirewall();
//Stop the port scanning defense before closing the app
psStopFirewall();
}

/*************************************************
Inputs: Character pointer formatted IP address,
Unsigned integer (8 bits) protocol
Unsigned integer (16 bits) port
Returns: None
Post: The IP is added to our list of IPs to block
*************************************************/
void PacketFilter::AddIPToBlockList(char* cIPAddr,
UINT8 proto,
UINT16 prt) {
//Check to make sure cIPAddr has some value (check validity later)
if(cIPAddr != NULL){
pFILTERINFO IPFilter = {0}; //Initialize the FilterInfo struct

/*Fill in the FilterInfo structure with our data*/
//Fill in the port number
        IPFilter.Port = prt;
//Fill in the protocol value
IPFilter.Protocol = proto;
//Get byte array format and hex format IP address from
//string format. Only add to the list if IP is valid
//and successfully formatted
if (FormatIPAddr(cIPAddr,
                         lstrlen(cIPAddr),
                         IPFilter.ByteAddr,
 IPFilter.HexAddr))
```

```
{
//Add the filter to the list
pFilterList.push_back(IPFilter);
}
}
}


/*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The packet filter is running and active
*************************************************/
int PacketFilter::pfStartFirewall(){
//Check if firewall is already running
if (pfrunning) {
//Indicate packet filter is already running
return PF_ALREADY_RUNNING;
}
else{
        //Create the packet filter interface
if (pfCreateInterface() == ERROR_SUCCESS){
            //Bind the packet filter interface
if (pfBindInterface() == ERROR_SUCCESS){
                //Add the filters
if (pfAddFilter() == ERROR_SUCCESS){
//Packet filter is now running
pfrunning = true;
//Message indicating packet filter is running
return PF_NOW_RUNNING;
}
            }
        }
    }
//Failed at some point in starting
//the packet filter
return PF_FAILED;
}
```

```
/************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The packet filter is stopped and inactive
************************************************/
int PacketFilter::pfStopFirewall(){
//If the packet filter is not running
if (!pfrunning) {
//Indicate the packet filter is already stopped
return PF_ALREADY_STOPPED;
}
else //If the packet filter is running
    {
        //Delete the filters
        pfDelFilter();
//Clear out the filter list
        pFilterList.clear();

        //Unbind from the packet filter interface
if(pfUnbindInterface() == ERROR_SUCCESS)
        {
            //Delete the packet filter interface
if(pfDeleteInterface() == ERROR_SUCCESS)
            {
                //Packet filter is no longer running
pfrunning = false;
//Return indicating packet filter is now stopped
return PF_NOW_STOPPED;
            }
        }
    }
//Failed at some point in stopping the packet filter
return PF_FAILED;
}

/************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The port scanning defense is running and active
************************************************/
```

```
int PacketFilter::psStartFirewall(){
//Check if port scanning defense is already running
if (psrunning) {
//Indicates port scanning defense is currently running
return PS_ALREADY_RUNNING;
}
else{
        //Create the port scanning defense interface
if (psCreateInterface() == ERROR_SUCCESS){
            //Bind the port scanning defense interface
if (psBindInterface() == ERROR_SUCCESS){
                //Add the filters
if (psAddFilter() == ERROR_SUCCESS){
//port scanning defense is now running
psrunning = true;
//Message indicating port scanning defense is running
return PS_NOW_RUNNING;
}
            }
        }
    }
//Failed at some point in starting
//the port scanning defense
return PS_FAILED;
}


/*************************************************
Inputs: None
Returns: Integer indicating running state (see Error Codes)
Post: The port scanning defense is stopped and inactive
*************************************************/
int PacketFilter::psStopFirewall(){
//If the port scanning defense is not running
if (!psrunning) {
//Message indicating port scanning defense already stopped
return PS_ALREADY_STOPPED;
}
else
    {
```

```
        //Delete the filters
        psDelFilter();

        //Unbind from the port scanning defense interface
if(psUnbindInterface() == ERROR_SUCCESS)
        {
            //Delete the port scanning defense interface
if(psDeleteInterface() == ERROR_SUCCESS)
            {
                //Port scanning defense is no longer running
psrunning = false;
//Message indicating port scanning defense is now stopped
return PS_NOW_STOPPED;
            }
        }
    }
//Failed at some point in stopping
//the port scanning defense
return PS_FAILED;
}


/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning Interface is created
*************************************************/
DWORD PacketFilter::psCreateInterface(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

    //Create port scanning defense interface by
//opening a session with the filtering engine
    ErrorCode =  FwpmEngineOpen0(//Must be NULL
 NULL,
 //Specifies authentication service to use
 RPC_C_AUTHN_WINNT,
 //Authorize credentials for filter engine (optional)
 NULL,
 //Session specific parameters (optional)
```

```
                                    NULL,
 //Engine handle set
                                    &psEngineHandle );

//Error code indicating any errors or success
return ErrorCode;
}

/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning Interface is deleted
*************************************************/
DWORD PacketFilter::psDeleteInterface(){
//ErrorCode for WFP functions, default is bad command
DWORD ErrorCode = ERROR_BAD_COMMAND;

//Check to see our Engine Handle is still correct
if(psEngineHandle != NULL){
//Close the interface
        ErrorCode = FwpmEngineClose0(psEngineHandle);
        psEngineHandle = NULL; //Reset the Engine Handle
    }
//Error code indicating any errors or success
    return ErrorCode;
}


/*************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning defense Interface is bound
*************************************************/
DWORD PacketFilter::psBindInterface(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;
//Remote Procedure Call status
    RPC_STATUS rpcStatus = {0};
//Sublayer data structure
    FWPM_SUBLAYER0 SubLayer = {0};
```

```
    //Create a GUID for the SubLayer
//for easy access and deletion
    rpcStatus = UuidCreate(&SubLayer.subLayerKey);

    if (rpcStatus == NO_ERROR){
/*Save the GUID for future use. Copies memory
**directly from SubLayer to pssubGUID from
**beginning of subLayerKey to the size of subLayerKey*/
        CopyMemory(&pssubGUID,
                   &SubLayer.subLayerKey,
                   sizeof(SubLayer.subLayerKey));

        //Fill in the SubLayer information
        SubLayer.displayData.name = PS_SUBLAYER_NAMEW;
        SubLayer.displayData.description = PS_SUBLAYER_NAMEW;
        SubLayer.flags = 0;
        SubLayer.weight = 0x100; //Importance of the sublayer

        //Add the SubLayer to the interface
        ErrorCode = FwpmSubLayerAdd0(psEngineHandle,
                                     &SubLayer,
                                     NULL);
}
//Error code indicating any errors or success
return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning defense Interface is unbound
************************************************/
DWORD PacketFilter::psUnbindInterface(){
//Delete the sublayer from the interface
    DWORD ErrorCode = FwpmSubLayerDeleteByKey0(psEngineHandle,
                                               &pssubGUID);
//Fill the GUID memory space with 0s
    ZeroMemory(&pssubGUID, sizeof(GUID));
```

```
//Error code indicating any errors or success
    return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning defense filters are added
  to our interface
************************************************/
DWORD PacketFilter::psAddFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;
//Constant number of conditions required
const UINT8 numCond = 1;
//Filter structure for filter info
FWPM_FILTER0 Filter;
//Fill filter structure with 0s
ZeroMemory(&Filter, sizeof(FWPM_FILTER));
//Filter condition structure for trigger condition info
    FWPM_FILTER_CONDITION0 Conditions;
//Fill filter condition structure with 0s
ZeroMemory(&Conditions, sizeof(FWPM_FILTER_CONDITION));
//Boolean to check if the transaction is in progress
bool txninProgress = FALSE;

/*Fill filter in with needed information*/
//Name of Filter for displaying only
Filter.displayData.name = PS_SUBLAYER_NAMEW;
//GUID for easy access and deletion
Filter.subLayerKey = pssubGUID;
//Number of conditions on our filter
Filter.numFilterConditions = numCond;
//Conditions to be fulfilled for action
//to be taken (previous condition struct)
Filter.filterCondition = &Conditions;

/*Since we're adding multiple conditions at once,
**use a transaction to clean up any partial additions
**in case of errors*/
```

```
ErrorCode = FwpmTransactionBegin0(psEngineHandle, 0);
txninProgress = TRUE;

/*Block outbound ICMP Destination Unreachable
to prevent UDP port scanning*/

Conditions.fieldKey = FWPM_CONDITION_ICMP_TYPE;
Conditions.matchType = FWP_MATCH_EQUAL;
Conditions.conditionValue.type = FWP_UINT16;
Conditions.conditionValue.uint16 = 3;

Filter.action.type = FWP_ACTION_BLOCK;

//Add the filter for V4
Filter.layerKey = FWPM_LAYER_OUTBOUND_ICMP_ERROR_V4;
ErrorCode = FwpmFilterAdd0(psEngineHandle,
   &Filter,
   NULL,
   &psFilterID[0]);

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

//Add the filter for V6
Filter.layerKey = FWPM_LAYER_OUTBOUND_ICMP_ERROR_V6;
ErrorCode = FwpmFilterAdd0(psEngineHandle,
   &Filter,
   NULL,
   &psFilterID[1]);

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
```

```
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

/*Block outbound TCP RST to prevent
**TCP port scanning*/

//Don't want to block RST from loopback, so create a
//condition that matches everything but loopback
Conditions.fieldKey = FWPM_CONDITION_FLAGS;
Conditions.matchType = FWP_MATCH_FLAGS_NONE_SET;
Conditions.conditionValue.type = FWP_UINT32;
Conditions.conditionValue.uint32 = FWP_CONDITION_FLAG_IS_LOOPBACK;

Filter.action.type = FWP_ACTION_CALLOUT_TERMINATING;

//Add the filter for IPv4 on the discard layer
Filter.layerKey = FWPM_LAYER_INBOUND_TRANSPORT_V4_DISCARD;
Filter.action.calloutKey =
FWPM_CALLOUT_WFP_TRANSPORT_LAYER_V4_SILENT_DROP;
ErrorCode = FwpmFilterAdd0(psEngineHandle,
   &Filter,
   NULL,
   &psFilterID[2]);

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

//Add the filter for IPv6 on the discard layer
Filter.layerKey = FWPM_LAYER_INBOUND_TRANSPORT_V6_DISCARD;
Filter.action.calloutKey =
FWPM_CALLOUT_WFP_TRANSPORT_LAYER_V6_SILENT_DROP;
ErrorCode = FwpmFilterAdd0(psEngineHandle,
   &Filter,
   NULL,
   &psFilterID[3]);
```

```
//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

//If everything succeeded, then we add the filters atomically
//At this point we are actually adding the filters
ErrorCode = FwpmTransactionCommit0(psEngineHandle);
txninProgress = FALSE;

//Cleanup operation if any filter additions failed
CLEANUP:
//if transaction is in progress and we had an error
//(from a goto) then abort the transaction (do nothing)
if ((txninProgress) && (ErrorCode != ERROR_SUCCESS)){
//Abort the transaction currently in progress
FwpmTransactionAbort0(psEngineHandle);
}

//Error code indicating any errors or success
return ErrorCode;
}


/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The port scanning defense filters are removed
  from our interface
************************************************/
DWORD PacketFilter::psDelFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

//For all 4 of our filters
for(int i=0; i<4; i++){
```

```
//Delete the filter via the ID (GUID)
ErrorCode = FwpmFilterDeleteById0(psEngineHandle,
                                          psFilterID[i]);
        //Reset the ID field
psFilterID[i] = 0;
}

//Error code indicating any errors or success
    return ErrorCode;
}



/*************************************************
Inputs: cIPAddr: String formatted IP address
nStrLen: Length of the IP address string
bIPAddr: Byte array for the IP address
hIPAddr: Unsigned long that contains the
 IP in hexadecimal format
Returns: True (IP formatted successfully)
     or False (Invalid IP)
Post: The FilterInfo contains the IP address
  in byte array and hexadecimal format
*************************************************/
bool PacketFilter::FormatIPAddr(char* cIPAddr,
UINT nStrLen,
BYTE* bIPAddr,
ULONG& hIPAddr) {
//The current character we are on
    UINT i = 0;
//The current octet we are on
    UINT j = 0;
//Stores the octet value
    UINT valOct = 0;
//Temporary storage for the current character
    char temp;

// Build byte array format from string format.
    for(; (i < nStrLen) && (j < 4); i++)
    {
```

```
//If we are still on the same octet
        if(cIPAddr[i] != '.')
        {
//Copy the character to temp
            temp = cIPAddr[i];
//Update the current octet value
valOct = (valOct*10) + (temp - '0');
        }
        else //If we are starting a new octet
        {
//Invalid octet value
if ((valOct < 0) || (valOct > 255)) return false;

//Store the octet value in the byte array
            bIPAddr[j] = valOct;
            valOct = 0;  //Reset the octet value
            j++;   //We are on a new octet
        }
    }
//IP has less then 4 octets (invalid IP address)
if (j != 3) return false;

//Store the last octect (we do not encounter
//another '.' at the end of the string)
    bIPAddr[j] = valOct;

//Format into hexadecimal from the byte array
    for(int k=0; k < 4; k++)
    {
//Use bitshifts to properly format the IP
        hIPAddr = (hIPAddr << 8) + bIPAddr[k];
    }

//IP address successfully formatted and valid
return true;
}


/**************************************************
Inputs: None
```

```
Returns: Error code indicating any errors
Post: The packet filter Interface is created
************************************************/
DWORD PacketFilter::pfCreateInterface(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

    //Create packet filter interface by opening
//a session with the filtering engine
    ErrorCode =  FwpmEngineOpen0(//Must be NULL
 NULL,
 //Specifies authentication service to use
 RPC_C_AUTHN_WINNT,
 //Authorize credentials for filter engine (optional)
 NULL,
 //Session specific parameters (optional)
                                  NULL,
 //Engine handle set
                                  &pfEngineHandle );

//Error code indicating any errors or success
return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The packet filter Interface is deleted
************************************************/
DWORD PacketFilter::pfDeleteInterface(){
//ErrorCode for WFP functions, default is bad command
DWORD ErrorCode = ERROR_BAD_COMMAND;

//Check to see our Engine Handle is still correct
if(pfEngineHandle != NULL){
//Close the interface
        ErrorCode = FwpmEngineClose0(pfEngineHandle);
        pfEngineHandle = NULL; //Reset the Engine Handle
    }
```

```
//Error code indicating any errors or success
    return ErrorCode;
}


/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The packet filter Interface is bound
************************************************/
DWORD PacketFilter::pfBindInterface(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;
//Remote Procedure Call status
    RPC_STATUS rpcStatus = {0};
//Sublayer structure
    FWPM_SUBLAYER0 SubLayer = {0};

    //Create a GUID for the SubLayer
//for easy access and deletion
    rpcStatus = UuidCreate(&SubLayer.subLayerKey);

    if (rpcStatus == NO_ERROR){
/*Save the GUID for future use. Copies memory
**directly from SubLayer to pssubGUID from
**beginning of subLayerKey to the size of subLayerKey*/
        CopyMemory(&pfsubGUID,
                   &SubLayer.subLayerKey,
                   sizeof(SubLayer.subLayerKey));
        //Fill in the SubLayer information
        SubLayer.displayData.name = PF_SUBLAYER_NAMEW;
        SubLayer.displayData.description = PF_SUBLAYER_NAMEW;
        SubLayer.flags = 0;
        SubLayer.weight = 0x100; //Importance of the sublayer

        //Add the SubLayer to the interface
        ErrorCode = FwpmSubLayerAdd0(pfEngineHandle,
                                     &SubLayer,
                                     NULL);
}
```

```
//Error code indicating any errors or success
return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The packet filter Interface is unbound
************************************************/
DWORD PacketFilter::pfUnbindInterface(){
//Delete the sublayer from the interface
    DWORD ErrorCode = FwpmSubLayerDeleteByKey0(pfEngineHandle,
                                               &pfsubGUID);
//Fill the GUID memory space with 0s
    ZeroMemory(&pfsubGUID, sizeof(GUID));

//Error code indicating any errors or success
    return ErrorCode;
}

/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is added to our interface for both
  outbound and inbound traffic
************************************************/
DWORD PacketFilter::pfAddFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

//Make sure we have filters to add
    if(pFilterList.size() > 0){
//Create an iterator to keep track
//of what filter we are on
pFILTERINFOLIST::iterator numFilter;

//For each filter in our list
        for(numFilter = pFilterList.begin();
numFilter != pFilterList.end();
```

```
numFilter++){

//Constant number of conditions required
const UINT8 numCond = 3;
//Filter structure for filter info
FWPM_FILTER0 Filter;
//Fill filter structure with 0s
ZeroMemory(&Filter, sizeof(FWPM_FILTER));
            //Filter condition structure for trigger
//condition info (1 for each condition)
FWPM_FILTER_CONDITION0 Conditions[numCond];
//Fill filter condition structures with 0s
ZeroMemory(&Conditions,
   sizeof(FWPM_FILTER_CONDITION) * numCond);
            //IP address and subnet mask data structure (v4)
FWP_V4_ADDR_AND_MASK AddrMask = {0};
//Boolean to check if the transaction is in progress
bool txninProgress = FALSE;

//Begin the transaction
ErrorCode = FwpmTransactionBegin0(pfEngineHandle, 0);
txninProgress = TRUE;

/*IP address info*/
//IP Address in Hexadecimal format
            AddrMask.addr = numFilter->HexAddr;
            //Subnet mask (default global subnet mask)
AddrMask.mask = SUBNET_MASK;

/*Fill condition in with needed information for IP address*/
//Check the remote address (where packet is coming from)
            Conditions[0].fieldKey =
FWPM_CONDITION_IP_REMOTE_ADDRESS;
//IP addresses should equal to trigger condition
            Conditions[0].matchType = FWP_MATCH_EQUAL;
//We are comparing IPv4 addresses and masks
            Conditions[0].conditionValue.type = FWP_V4_ADDR_MASK;
//IP address and subnet mask we are checking for
            Conditions[0].conditionValue.v4AddrMask = &AddrMask;
```

```
/*Fill condition in with needed information for Protocol*/
//Check the protocol
Conditions[1].fieldKey = FWPM_CONDITION_IP_PROTOCOL;
//Protocol should be the same to trigger condition
Conditions[1].matchType = FWP_MATCH_EQUAL;
//Protocol is an unsigned Integer of 8 bits
Conditions[1].conditionValue.type = FWP_UINT8;
            //Protocol we want to block is in numFilter
Conditions[1].conditionValue.uint32 =
numFilter->Protocol;

/*Fill condition in with needed information for Port*/

//Check the incoming port number
Conditions[2].fieldKey = FWPM_CONDITION_IP_REMOTE_PORT;
//0 indicates we want to block all ports
if ((numFilter->Port) == 0) {
//All ports greater than or
//equal to 0 (in other words, all)
Conditions[2].matchType = FWP_MATCH_GREATER_OR_EQUAL;
}
//We want to block a specific port
else {
//Match the specified port
Conditions[2].matchType = FWP_MATCH_EQUAL;
}
//Ports are unsigned Integers of 16 bits
Conditions[2].conditionValue.type = FWP_UINT16;
//Port we want to block is in numFilter
Conditions[2].conditionValue.uint16 = numFilter->Port;

            /*Fill filter in with needed information*/
//GUID for easy access
            Filter.subLayerKey = pfsubGUID;
//Name of Filter for displaying only
            Filter.displayData.name = PF_SUBLAYER_NAMEW;
//Inspect incoming IPv4 packets
            Filter.layerKey = FWPM_LAYER_INBOUND_TRANSPORT_V4;
//Block packets that meet our conditions
            Filter.action.type = FWP_ACTION_BLOCK;
```

```
//Let the engine auto assign the weight
            Filter.weight.type = FWP_EMPTY;
//Conditions to be fulfilled for action to be taken
            Filter.filterCondition = Conditions;
//Number of conditions on our filter
            Filter.numFilterConditions = numCond;

//Add the filter for inbound traffic
            ErrorCode = FwpmFilterAdd0(pfEngineHandle,
                                       &Filter,
                                       NULL,
                                       &(numFilter->InFilterID));

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

//Change the layer to the outbound layer
Filter.layerKey = FWPM_LAYER_OUTBOUND_TRANSPORT_V4;
//Add the filter for outbound

            ErrorCode = FwpmFilterAdd0(pfEngineHandle,
                                       &Filter,
                                       NULL,
                                       &(numFilter->OutFilterID));

//If not a success, begin cleanup
/*NOTE: goto in this instance, and all other instances,
**is used purely to prevent the code from being clogged
**up with multiple nested if blocks or the creation of
**otherwise unneeded variables. It is merely the easiest
**and most efficient way to preform the transaction*/
if (ErrorCode != ERROR_SUCCESS) goto CLEANUP;

//Commit the transaction
ErrorCode = FwpmTransactionCommit0(pfEngineHandle);
```

```
txninProgress = FALSE;

//Cleanup operation if any filter additions failed
CLEANUP:
//if transaction is in progress and we had an error
//(from a goto) then abort the transaction (do nothing)
if ((txninProgress) && (ErrorCode != ERROR_SUCCESS)) {
FwpmTransactionAbort0(pfEngineHandle);
}
}
}

//Error code indicating any errors or success
return ErrorCode;
}


/************************************************
Inputs: None
Returns: Error code indicating any errors
Post: The Filter is removed from our interface
************************************************/
DWORD PacketFilter::pfDelFilter(){
//ErrorCode for WFP functions, default is bad command
    DWORD ErrorCode = ERROR_BAD_COMMAND;

//Make sure our list is not empty
if(pFilterList.size() > 0){
//Create an iterator to go through our list
pFILTERINFOLIST::iterator numFilter;
//For each filter in the list
for(numFilter = pFilterList.begin();
numFilter != pFilterList.end();
numFilter++) {
//Delete the inbound traffic filter
ErrorCode = FwpmFilterDeleteById0(pfEngineHandle,
  numFilter->InFilterID);
//Delete the outbound traffic filter
if (ErrorCode == ERROR_SUCCESS) {
ErrorCode = FwpmFilterDeleteById0(pfEngineHandle,
```

```
  numFilter->OutFilterID);
numFilter->OutFilterID = 0;
}
            //Reset the ID
numFilter->InFilterID = 0;
//If an error is encoutered, stop
//trying to delete filters and report it
if (ErrorCode != ERROR_SUCCESS) break;
}
}

//Error code indicating any errors or success
    return ErrorCode;
}
```

# Appendix E

# Form1.h Code

```
/*********************************************
Created by Terry Rogers on September 4, 2010

Form1 Class
This file handles all GUI functions and data
*********************************************/



/********Includes********/
#include "Packetfilter.h"
#include "Appfilter.h"

/********Structures********/
//Structure to store inputted IP addresses
//until we start the firewall
typedef struct _PFBlocked {
char *IP;
UINT16 Port;
UINT8 Protocol;
} PFBlocked, *PPFBlocked;

//List of inputted IP addresses
typedef std::list<PFBlocked> PFBlockedList;

//Structure to store selected applications to block
typedef struct _APPBlocked {
```

```
char *appBlock;
unsigned int length;
} APPBlocked, *PAPPBlocked;

//List of selected applications
typedef std::list<APPBlocked> APPBlockedList;

/********Global Variables********/
//List of selected applications to block
APPBlockedList appList;

//List of inputted IP addresses to block
PFBlockedList pfList;

//Packet Filter
PacketFilter pf;

//Application Filter
AppFilter af;

#pragma once


namespace DinsFirewall {

using namespace System;
using namespace System::IO;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;

/// <summary>
/// Summary for Form1
///
/// WARNING: If you change the name of this class,
///  you will need to change the 'Resource File Name'
///          property for the managed resource compiler tool
///          associated with all .resx files this class depends on.
```

```
///  Otherwise, the designers will not be able to interact
///  properly with localized resources associated with this form.
/// </summary>


public ref class Form1 : public System::Windows::Forms::Form
{
public:
Form1(void)
{
InitializeComponent();
//
//TODO: Add the constructor code here
//
}

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
{
if (components)
{
delete components;
}
}
private: System::Windows::Forms::Button^  start;
protected:
private: System::Windows::Forms::Button^  stop;
private: System::Windows::Forms::TextBox^  txtPfStat;


private: System::Windows::Forms::Label^  lblStat;
private: System::Windows::Forms::Button^  btnAddBlk;
private: System::Windows::Forms::TextBox^  txtNewIP;
private: System::Windows::Forms::Button^  btnAppGather;

private: System::Windows::Forms::Label^  lblApp;
private: System::Windows::Forms::CheckedListBox^  chklistApps;
```

```
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::TextBox^  txtPort;
private: System::Windows::Forms::ComboBox^  comboPrtcl;
private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Label^  label3;
private: System::Windows::Forms::Label^  label4;
private: System::Windows::Forms::TextBox^  txtAppStat;
private: System::Windows::Forms::Button^  btnCurFile;
private: System::Windows::Forms::Label^  label5;
private: System::Windows::Forms::TextBox^  txtPsStat;


private: System::Windows::Forms::CheckBox^  chkPS;



private:
/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
System::ComponentModel::
ComponentResourceManager^  resources = (gcnew System::
ComponentModel::ComponentResourceManager(Form1::typeid));
this->start = (gcnew System::Windows::Forms::Button());
this->stop = (gcnew System::Windows::Forms::Button());
this->txtPfStat = (gcnew System::Windows::Forms::TextBox());
this->lblStat = (gcnew System::Windows::Forms::Label());
this->btnAddBlk = (gcnew System::Windows::Forms::Button());
this->txtNewIP = (gcnew System::Windows::Forms::TextBox());
this->btnAppGather = (gcnew System::Windows::Forms::Button());
this->lblApp = (gcnew System::Windows::Forms::Label());
```

```
this->chklistApps = (gcnew System::Windows::Forms::
CheckedListBox());
this->label1 = (gcnew System::Windows::Forms::Label());
this->txtPort = (gcnew System::Windows::Forms::TextBox());
this->comboPrtcl = (gcnew System::Windows::Forms::ComboBox());
this->label2 = (gcnew System::Windows::Forms::Label());
this->label3 = (gcnew System::Windows::Forms::Label());
this->label4 = (gcnew System::Windows::Forms::Label());
this->txtAppStat = (gcnew System::Windows::Forms::TextBox());
this->btnCurFile = (gcnew System::Windows::Forms::Button());
this->label5 = (gcnew System::Windows::Forms::Label());
this->txtPsStat = (gcnew System::Windows::Forms::TextBox());
this->chkPS = (gcnew System::Windows::Forms::CheckBox());
this->SuspendLayout();
//
// start
//
this->start->Location = System::Drawing::Point(12, 306);
this->start->Name = L"start";
this->start->Size = System::Drawing::Size(75, 23);
this->start->TabIndex = 7;
this->start->Text = L"Start Firewall";
this->start->UseVisualStyleBackColor = true;
this->start->Click += gcnew System::
EventHandler(this, &Form1::start_Click);
//
// stop
//
this->stop->Location = System::Drawing::Point(192, 306);
this->stop->Name = L"stop";
this->stop->Size = System::Drawing::Size(75, 23);
this->stop->TabIndex = 8;
this->stop->Text = L"Stop Firewall";
this->stop->UseVisualStyleBackColor = true;
this->stop->Click += gcnew System::
EventHandler(this, &Form1::stop_Click);
//
// txtPfStat
//
this->txtPfStat->BorderStyle =
```

```
System::Windows::Forms::BorderStyle::FixedSingle;
this->txtPfStat->Location = System::Drawing::Point(96, 9);
this->txtPfStat->Name = L"txtPfStat";
this->txtPfStat->ReadOnly = true;
this->txtPfStat->Size = System::Drawing::Size(100, 20);
this->txtPfStat->TabIndex = 2;
this->txtPfStat->TabStop = false;
this->txtPfStat->Text = L"Now Stopped";
//
// lblStat
//
this->lblStat->AutoSize = true;
this->lblStat->Location = System::Drawing::Point(9, 9);
this->lblStat->Name = L"lblStat";
this->lblStat->Size = System::Drawing::Size(81, 13);
this->lblStat->TabIndex = 3;
this->lblStat->Text = L"PF Filter Status:";
//
// btnAddBlk
//
this->btnAddBlk->Location = System::Drawing::Point(192, 156);
this->btnAddBlk->Name = L"btnAddBlk";
this->btnAddBlk->Size = System::Drawing::Size(60, 46);
this->btnAddBlk->TabIndex = 3;
this->btnAddBlk->Text = L"Add To Blocked";
this->btnAddBlk->UseVisualStyleBackColor = true;
this->btnAddBlk->Click += gcnew System::
EventHandler(this, &Form1::btnAddBlk_Click);
//
// txtNewIP
//
this->txtNewIP->Location = System::Drawing::Point(76, 105);
this->txtNewIP->Name = L"txtNewIP";
this->txtNewIP->Size = System::Drawing::Size(97, 20);
this->txtNewIP->TabIndex = 0;
//
// btnAppGather
//
this->btnAppGather->Location = System::Drawing::Point(382, 27);
this->btnAppGather->Name = L"btnAppGather";
```

```
this->btnAppGather->Size = System::Drawing::Size(101, 40);
this->btnAppGather->TabIndex = 4;
this->btnAppGather->Text = L"Begin Filepath Gathering";
this->btnAppGather->UseVisualStyleBackColor = true;
this->btnAppGather->Click += gcnew System::
EventHandler(this, &Form1::btnAppGather_Click);
//
// lblApp
//
this->lblApp->AutoSize = true;
this->lblApp->Location = System::Drawing::Point(379, 9);
this->lblApp->Name = L"lblApp";
this->lblApp->Size = System::Drawing::Size(118, 13);
this->lblApp->TabIndex = 7;
this->lblApp->Text = L"Not Gathering Filepaths";
//
// chklistApps
//
this->chklistApps->CheckOnClick = true;
this->chklistApps->FormattingEnabled = true;
this->chklistApps->HorizontalScrollbar = true;
this->chklistApps->Location = System::Drawing::Point(298, 135);
this->chklistApps->Name = L"chklistApps";
this->chklistApps->Size = System::Drawing::Size(246, 184);
this->chklistApps->Sorted = true;
this->chklistApps->TabIndex = 6;
//
// label1
//
this->label1->AutoSize = true;
this->label1->Location = System::Drawing::Point(12, 108);
this->label1->Name = L"label1";
this->label1->Size = System::Drawing::Size(58, 13);
this->label1->TabIndex = 9;
this->label1->Text = L"IP Address";
//
// txtPort
//
this->txtPort->Location = System::Drawing::Point(77, 140);
this->txtPort->Name = L"txtPort";
```

```
this->txtPort->Size = System::Drawing::Size(95, 20);
this->txtPort->TabIndex = 1;
//
// comboPrtcl
//
this->comboPrtcl->AllowDrop = true;
this->comboPrtcl->FormattingEnabled = true;
this->comboPrtcl->Items->AddRange
(gcnew cli::array< System::Object^  >(7) {
L"TCP",
L"UDP",
L"ICMP",
L"ICMP_6",
L"IGMP",
L"BLUET",
L"RM"});
this->comboPrtcl->Location = System::Drawing::Point(76, 181);
this->comboPrtcl->Name = L"comboPrtcl";
this->comboPrtcl->Size = System::Drawing::Size(102, 21);
this->comboPrtcl->TabIndex = 2;
this->comboPrtcl->Text = L"Select a Protocol";
//
// label2
//
this->label2->AutoSize = true;
this->label2->Location = System::Drawing::Point(12, 184);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(51, 13);
this->label2->TabIndex = 12;
this->label2->Text = L"Protocols";
//
// label3
//
this->label3->AutoSize = true;
this->label3->Location = System::Drawing::Point(12, 143);
this->label3->Name = L"label3";
this->label3->Size = System::Drawing::Size(26, 13);
this->label3->TabIndex = 13;
this->label3->Text = L"Port";
//
```

```
// label4
//
this->label4->AutoSize = true;
this->label4->Location = System::Drawing::Point(9, 41);
this->label4->Name = L"label4";
this->label4->Size = System::Drawing::Size(84, 13);
this->label4->TabIndex = 14;
this->label4->Text = L"AppFilter Status:";
//
// txtAppStat
//
this->txtAppStat->BorderStyle =
System::Windows::Forms::BorderStyle::FixedSingle;
this->txtAppStat->Location = System::Drawing::Point(96, 38);
this->txtAppStat->Name = L"txtAppStat";
this->txtAppStat->ReadOnly = true;
this->txtAppStat->Size = System::Drawing::Size(100, 20);
this->txtAppStat->TabIndex = 15;
this->txtAppStat->TabStop = false;
this->txtAppStat->Text = L"Now Stopped";
//
// btnCurFile
//
this->btnCurFile->Location = System::Drawing::Point(384, 81);
this->btnCurFile->Name = L"btnCurFile";
this->btnCurFile->Size = System::Drawing::Size(98, 30);
this->btnCurFile->TabIndex = 5;
this->btnCurFile->Text = L"Use current file";
this->btnCurFile->UseVisualStyleBackColor = true;
this->btnCurFile->Click += gcnew System::
EventHandler(this, &Form1::btnCurFile_Click);
//
// label5
//
this->label5->AutoSize = true;
this->label5->Location = System::Drawing::Point(9, 70);
this->label5->Name = L"label5";
this->label5->Size = System::Drawing::Size(87, 13);
this->label5->TabIndex = 16;
this->label5->Text = L"PortScan Status:";
```

```
//
// txtPsStat
//
this->txtPsStat->BorderStyle =
System::Windows::Forms::BorderStyle::FixedSingle;
this->txtPsStat->Location = System::Drawing::Point(96, 67);
this->txtPsStat->Name = L"txtPsStat";
this->txtPsStat->ReadOnly = true;
this->txtPsStat->Size = System::Drawing::Size(100, 20);
this->txtPsStat->TabIndex = 17;
this->txtPsStat->Text = L"Now Stopped";
//
// chkPS
//
this->chkPS->AutoSize = true;
this->chkPS->Location = System::Drawing::Point(12, 225);
this->chkPS->Name = L"chkPS";
this->chkPS->Size = System::Drawing::Size(142, 17);
this->chkPS->TabIndex = 18;
this->chkPS->Text = L"Port Scanning Defense\?";
this->chkPS->UseVisualStyleBackColor = true;
//
// Form1
//
this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
this->AutoScaleMode = System::
Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(569, 384);
this->Controls->Add(this->chkPS);
this->Controls->Add(this->txtPsStat);
this->Controls->Add(this->label5);
this->Controls->Add(this->btnCurFile);
this->Controls->Add(this->txtAppStat);
this->Controls->Add(this->label4);
this->Controls->Add(this->label3);
this->Controls->Add(this->label2);
this->Controls->Add(this->comboPrtcl);
this->Controls->Add(this->txtPort);
this->Controls->Add(this->label1);
this->Controls->Add(this->chklistApps);
```

```
this->Controls->Add(this->lblApp);
this->Controls->Add(this->btnAppGather);
this->Controls->Add(this->txtNewIP);
this->Controls->Add(this->btnAddBlk);
this->Controls->Add(this->lblStat);
this->Controls->Add(this->txtPfStat);
this->Controls->Add(this->stop);
this->Controls->Add(this->start);
this->Icon = (cli::safe_cast<System::Drawing::Icon^  >
(resources->GetObject(L"$this.Icon")));
this->Name = L"Form1";
this->Text = L"Din\'s Firewall";
this->ResumeLayout(false);
this->PerformLayout();

}
#pragma endregion

 /*Start the Packet Filter and Application firewalls,
 if applicable, when "Start" button is clicked*/
private: System::Void start_Click
 (System::Object^  sender, System::EventArgs^  e) {
 //Build the list of applications based on checked items
 APPBlockBuilder();

 //Add all the IPs to the block list
 PFBlockedList::iterator PfnumFilter;
 for(PfnumFilter = pfList.begin();
 PfnumFilter != pfList.end();
 PfnumFilter++){
pf.AddIPToBlockList(PfnumFilter->IP,
PfnumFilter->Protocol,
PfnumFilter->Port);
 }

 //Add all apps to the block list
   APPBlockedList::iterator AppnumFilter;
         for(AppnumFilter = appList.begin();
 AppnumFilter != appList.end();
 AppnumFilter++){
```

```
af.AddAppToBlockList(AppnumFilter->appBlock,
 AppnumFilter->length);
 }

 //Check if port scanning defense is desired
 //If it is checked, start it
 if (chkPS->Checked) {
 int psresult = pf.psStartFirewall();
 //Update the display based on any errors
 if (psresult == PS_ALREADY_RUNNING)
 txtPsStat->Text = "Already running";
 else {
 if (psresult == PS_NOW_RUNNING)
 txtPsStat->Text = "Now running";
 else
 txtPsStat->Text = "Error!";
 }
 }
 //Else, update the display to indicate it is not running
 else txtPsStat->Text = "Not Running";

 //Check if any IPs have been added to be blocked
 if (!pfList.empty()){
 int pfresult = pf.pfStartFirewall();
 //Update the display based on any errors
 if (pfresult == PF_ALREADY_RUNNING)
 txtPfStat->Text = "Already running";
 else {
 if (pfresult == PF_NOW_RUNNING)
 txtPfStat->Text = "Now running";
 else
 txtPfStat->Text = "Error!";
 }
 }
 //Else, update the display to indicate it is not running
 else txtPfStat->Text = "Not Running";

 //Check if any applications were selected
 if (!appList.empty()){
 //Start the Application Firewall
```

```
 int appresult = af.appStartFirewall();
 //Update the display based on any errors
 if (appresult == APP_ALREADY_RUNNING)
 txtAppStat->Text = "Already running";
 else {
 if (appresult == APP_NOW_RUNNING)
 txtAppStat->Text = "Now running";
 else
 txtAppStat->Text = "Error!";
 }
 }
 //Else, update the display to indicate it is not running
 else txtAppStat->Text = "Not Running";
}

 /*Stop the Port Scanning Defense,
 Packet Filter, and Application Filter
 when the "Stop" button is clicked*/
private: System::Void stop_Click
 (System::Object^  sender, System::EventArgs^  e) {
 //Stop the Port Scanning Defense
 int psresult = pf.psStopFirewall();

 //Update the display based on any errors
 if (psresult == PS_ALREADY_STOPPED)
txtPsStat->Text = "Already stopped";
else {
if (psresult == PS_NOW_STOPPED)
txtPsStat->Text = "Now stopped";
else
txtPsStat->Text = "Error!";
 }

 //Stop the Packet Filter Firewall
 int pfresult = pf.pfStopFirewall();

 //Clear our list of blocked IPs
    pfList.clear();

 //Update the display based on any errors
```

```
 if (pfresult == PF_ALREADY_STOPPED)
 txtPfStat->Text = "Already Stopped";
 else {
 if (pfresult == PF_NOW_STOPPED)
txtPfStat->Text = "Now Stopped";
 else
txtPfStat->Text = "Error!";
 }

 //Stop the Application Firewall
 int appresult = af.appStopFirewall();

 //Clear our list of blocked apps
    appList.clear();

 //Update the display based on any errors
 if (appresult == APP_ALREADY_STOPPED)
 txtAppStat->Text = "Already Stopped";
 else {
 if (appresult == APP_NOW_STOPPED)
txtAppStat->Text = "Now Stopped";
 else
txtAppStat->Text = "Error!";
 }
 }

 /*Add an IP address to the list of IPs to
 block when the "Add To Blocked" button is clicked*/
private: System::Void btnAddBlk_Click
 (System::Object^  sender, System::EventArgs^  e) {
//Make sure the packet filter is not
//running before adding new IPs
if (!pf.psrunning) {

//Create a new data structure to store the info
PFBlocked Blocked = {0};

//Format the IP address to character
//pointer, add to storage structure
Blocked.IP = (char*)(System::Runtime::InteropServices::Marshal::
```

```
StringToHGlobalAnsi(txtNewIP->Text)).ToPointer();

//Format the Port number, add to storage structure
short port;
Int16::TryParse(txtPort->Text, port);
Blocked.Port = (UINT16)port;

//Cases depending on which combo box item is
//selected for the protocol
switch (comboPrtcl->SelectedIndex) {
case 0: //TCP
Blocked.Protocol = TCP; //Protocol is TCP
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 1: //UDP
Blocked.Protocol = UDP; //Protocol is UDP
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 2: //ICMP
Blocked.Protocol = ICMP; //Protocol is ICMP
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 3: //ICMP Version 6
Blocked.Protocol = ICMP_6; //Protocol is ICMP V6
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 4: //IGMP
Blocked.Protocol = IGMP; //Protocol is IGMP
```

```
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 5: //Bluetooth
Blocked.Protocol = BLUET; //Protocol is Bluetooth
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
case 6: //RM
Blocked.Protocol = RM; //Protocol is RM
pfList.push_back(Blocked); //Add the structure to the list
txtPort->Text = ""; //Clear the port text box
txtNewIP->Text = ""; //Clear the IP text box
txtNewIP->Focus(); //Put cursor back in IP text box
break;
default: //Error or no protocol selected
MessageBox::Show("Please select a protocol!");
}
}
else { //Cannot add IPs while filter is already running
MessageBox::Show("You must stop the firewall " +
"before adding new IPs");
//Have the "Stop" button selected (bring into focus)
stop->Focus();
}
}

/*Build a list of all applications selected to be blocked*/
private: void APPBlockBuilder() {

//Get a list of all checked items
IEnumerator^ Enum = chklistApps->CheckedItems->GetEnumerator();

//Go through each checked item
while (Enum->MoveNext()){
//Current checked object we are on
```

```
Object^ Checked = safe_cast<Object^>(Enum->Current);

//Filepath of the checked object
String^ path = gcnew String(Checked->ToString());
//Add a null char to indicate end of string
path = String::Concat(path, "\0");

//Create a structure to store the data
APPBlocked Blocked = {0};

//Get the length of the filepath for later type conversion
Blocked.length = path->Length;

//Store the filepath (converted to character pointer)
//into the structure
Blocked.appBlock = (char*)(System::Runtime::InteropServices::
Marshal::StringToHGlobalAnsi(path)).ToPointer();

//Add the structure to the list of blocked applications
appList.push_back(Blocked);
}
}
 /*Gather Filepaths of all Applications (all .exe)*/
private: System::Void btnAppGather_Click
 (System::Object^  sender, System::EventArgs^  e) {

//Clear the box before we populate it with new filepaths
chklistApps->Items->Clear();

//If the filepath file exists, delete it to
//make room for our new, more current one.
if (File::Exists("AppList.txt")) {
File::Delete("AppList.txt");
}

//Search recursively for Applications in the C drive
AppSearch("C:\\");
//Update label to indicate scan completed
lblApp->Text = "Scan complete";
}
```

```
/*Recursively search through directories to find all .exe files*/
private: void AppSearch(String^ sDir){
//Error code for getting attributes
        DWORD result;

//Character pointer formatted filepath
//to gather folder attributes
char* attributes;

try{
 //Find all the subfolders in the folder that is passed in
 array<String^>^ dirs = Directory::GetDirectories(sDir);

 //Get the total number of subfolders
 int numDir = dirs->GetLength(0);

 for (int i=0; i<numDir; i++){ //For each subfolder
 //Format the folder path into a character pointer
 attributes = (char*)(System::Runtime::InteropServices::
 Marshal::StringToHGlobalAnsi(dirs[i])).ToPointer();
 //Get the attributes of the folder
 result = GetFileAttributes(attributes);

 //While the subfolder is a junction point
 //or is the System Volume Information folder
 while (((result & 0x400) == 0x400) ||
   (dirs[i]->Contains("System Volume"))) {
i++; //Move on to the next folder (skip it)

if (i<numDir) { //If there are still more subfolders
//Format the folder path into a character pointer
attributes = (char*)(System::Runtime::InteropServices::
Marshal::StringToHGlobalAnsi(dirs[i])).ToPointer();
//Get the attributes of the folder
result = GetFileAttributes(attributes);
}
else //If not, break out of the while loop
break;
 }
```

```
 if (i<numDir) { //If there are still more subfolders
 //Get all files that end in .exe
 array<String^>^ files = Directory::GetFiles(dirs[i],"*.exe");

 //Get the number of files found in the directory
 int numFile = files->GetLength(0);

 //For each file that has a .exe extension
 for (int j=0; j<numFile; j++){

 //Open the text file (created if it does not exist)
 StreamWriter^ writer = File::AppendText("AppList.txt");

 //Append the filepath to the text file on a new line
 writer->WriteLine(files[j]);

 //Add the file to the list on the GUI unselected
 chklistApps->Items->Add(files[j], CheckState::Unchecked);

 //Close the file writer
 writer->Close();
             }
 //Begin search on next folder, if there is one
 AppSearch(dirs[i]);
 }
 }
}
//If an exception (error) occurs
catch (System::Exception^ e)
{
//Display the exception (error)
MessageBox::Show(e->Message);
}
}

 /*Load up a previously created list of applications if
 the "Use Current File" button is clicked*/
private: System::Void btnCurFile_Click
 (System::Object^  sender, System::EventArgs^  e) {
```

```
 //Clear the list before we populate it with new entries
 chklistApps->Items->Clear();

 //Open the file for reading
 StreamReader^ reader = gcnew StreamReader("AppList.txt");

 try {
 String^ line; //String for each line of text

 //While there is still a valid line
 while (line = reader->ReadLine() ){
 //Add the line to the box
 chklistApps->Items->Add(line, CheckState::Unchecked);
 }
 }

 catch (System::Exception^ e) { //If an exception (error) occurs
 MessageBox::Show(e->Message); //Display the exception (error)
 }

 //Close the file
 reader->Close();
 //Update label display
 lblApp->Text = "File loading complete";
 }
};
}
```