# Project: Stateful Firewalls

Project originally created by: Peter A. H. Peterson and Dr. Peter Reiher, UCLA {pahp, reiher}@cs.ucla.edu, with Dr. Tanya Crenshaw, UP [{crenshaw@up.edu](mailto:%257Bcrenshaw@up.edu)}

Modified by Prem Uppuluri (Radford University). Modifications pertain to limiting this exercise to firewall configuration and changing the firewall configuration.

#### Tasks

A **firewall.sh** script is provided to you on D2L. This contains some very basic IP table rules to get you started. You can add your rules to this shell program and whenever you are done, run it as follows:

$ ./firewall.sh

### Tasks and Deliverables

**The firewall is** not **enabled by default -- to enforce the rules, execute the following two commands on the server:**

1) iptables --flush

2) ./firewall.sh

... as the root user or using sudo:

1. sudo /sbin/iptables --flush
2. sudo firewall.sh

This will load the rules and start enforcing them. **To make sure that you are removing all iptables rules**, you should run iptables “--flush”  
 in **between every invocation of firewall.sh** or rules might "stick around" which can be very confusing if you are trying to debug the system. This can be done like this as root (or with sudo as above).

**Here's what the firewall needs to do:**

1. **passively ignore** any traffic inbound to the interface that says it's coming from the server itself (obvious spoof attempt). The server uses the localhost loopback device lo for internal traffic, so it should never see incoming traffic from its own IP on the experimental network interface. (See test case 11, below)
2. Allow all *established* traffic on the experimental network interface. Established or related traffic is traffic that is part of *previously accepted* new connections.
3. Accept **new** connections on the experimental network (10.1.x.x) of the types listed below:
   1. Inbound TCP connections to the OpenSSH, Apache, and MySQL servers on their standard ports. (Test cases 1, 3, 5)
      * The MySQL server should **only** accept connections from the **client** host.
   2. Inbound UDP connections to the **server** ports 10000 to 10005 from the host **client**. (Test case 8)
   3. Inbound ICMP ping requests and replies. (Test case 6)
   4. Outbound TCP connections to any OpenSSH, SMTP, and Apache (on standard ports). (Test cases 2, 4)
   5. Outbound UDP connections to the ports 10006 to 10010 on host **client** from the **server**. (Test case 9)
   6. Outbound ICMP ping requests and replies. (Test case 7)
4. **passively ignore all other traffic**. (Do not allow it or respond to it in any way.) (Test case 10)

There are many online resources and tutorials for iptables configuration -- feel free to use them. Be aware, however, **not all tutorials emphasize the principle of least privilege** and may give you overly permissive advice! In order to properly configure the firewall, you must consider the basic ways the firewall can differentiate traffic and allow only the specific types you require to properly function.

Please read the helpful advice in the next section for configuring and testing your firewall.

#### How to test your firewall

Testing your firewall is easy; you just need to make sure that the allowed services are allowed, and that things that should be denied are denied. To do that, we'll use a few tools like telnet, netcat, and others.

You may also have noticed that this experiment swaps in two nodes instead of one. One will be called client and the other will be called server. server is the node with the firewall and resources you want to protect, but you can use client to check to see if the firewall is doing its job. You can also use client as a target to see if the server's outbound rules are functioning properly, using tools such as nmap, telnet, nc, and others in the [network tools portion of this document](#nettools).

## Submission Instructions

Submit the firewall.sh file on D2L drop box.

## Required Reading

## Firewalls

### Stateless Firewalls

In the late 1980s, the Internet was just beginning to grow beyond its early academic and governmental applications into the commercial and personal worlds. The [Great Internet Worm](http://en.wikipedia.org/wiki/Morris_worm) in November of 1988 infected around 6,000 hosts (roughly 10% of the Internet) in the first major infection of its kind and helped to focus research and awareness on securing computers from unauthorized access. It was in this environment that the first firewalls were written about and developed at Digital Equipment Corporation (DEC) and Bell Labs (AT&T).

The first functional firewalls inspected individual packet headers without regard for established connections, other packets, or their contents. These kind of firewalls became known as "packet filters" because they literally filtered the packets one by one according to a set of criteria, not unlike a quality control inspector on an assembly line. For TCP and UDP, these criteria could be reduced essentially to the source and destination addresses and ports in the packet header. For example, a packet filter could reject or drop any packets destined for port 23 (telnet) on host 10.10.10.10 from any address other than 10.10.10.11. This kind of filter could rapidly and inexpensively inspect and classify packets without using much space (although they were not very "smart").

Unsurprisingly, simple packet filters are not adequate for many applications, such as the [File Transfer Protocol](http://en.wikipedia.org/wiki/Ftp) (FTP), because these protocols open additional connections on random ports that can not be anticipated or recognized by the firewall since it does not understand or consider the state of any connection.

This kind of simple "packet filter" ultimately became known as a "stateless firewall".

### Stateful Firewalls

"Stateful firewalls" arrived not long after "stateless firewalls". Stateful firewalls keep tables of network connections and states in memory in order to determine if a packet is part of a preexisting network connection, the start of a new and legitimate connection, or an unwanted or unrelated packet. This kind of firewall can recognize, for example, that a new connection on a random high port from a host with a preexisting FTP connection is a related connection and should be allowed. Another difference is that while a stateless firewall will allow all packets from acceptable hosts to an open port, a stateful firewall can be configured to allow packets to that port only if a legitimate TCP connection (or some other protocol) has already been established in some acceptable way. Understanding protocol state essentially gives stateful firewalls vastly more criteria in deciding whether to accept or reject a packet, which translates into finer granularity.

The cutting edge of firewall design today is what is called an "application-layer firewall", which is a firewall that performs "deep packet inspection". This means that the firewall is capable of looking not just at the header of the packets and the state of the connection, but at the payload of the packet in context of what the application processing the packets will do. For example, an application-layer firewall could be used to block Java applets from HTTP traffic by inspecting the packets and stripping Java code or dropping the packets entirely. In order to do this, it must understand what applet code looks like within the payload portion of any HTTP traffic stream. An application-layer firewall essentially has total control over the network stream, although this control comes at a significant expense in terms of CPU time and software complexity.

Most firewalls in use today lie somewhere between the stateful firewall and the application-layer firewall. These firewalls function essentially as a stateful firewall, but may understand enough of a few applications to perform some application-layer tasks. It is also common to couple a primarily stateful firewall (such as netfilter/iptables) with separate application layer firewalls for individual applications.

### Firewall Policy Design

People imagine many different things when they hear the term "[firewall](http://en.wikipedia.org/wiki/Firewall)" in the context of computer networking. Some envision an impenetrable wall of flame *[at least I did --ed.]*. A Hollywood screenwriter might envision Harrison Ford battling kidnappers. A mechanic might envision the wall between the engine and passenger compartment of a car. Yet mysteriously, every firewall is illustrated as a boring, red brick wall, typically with no fire in sight.

Actually, the brick wall isn't that strange -- the name "firewall" comes from the brick walls in buildings placed to stop the spread of a fire from one area to another. But no matter who you are or what you see in your minds eye, the conventional wisdom is that firewalls are used to "keep the bad stuff out," whether you're protecting your desktop PC at home, your office LAN, or the Pentagon. However, those of us in the field of computer security often see firewalls more as a means of keeping things *in* rather than keeping them *out*.

In one sense, these are two sides of the same coin -- but how you design something is (often unconsciously) directly related to how you view the problem, and this can lead to very different design choices when developing a firewall. The goal of "keeping things out" is by definition, exclusively concerned with keeping external attackers "outside" the system, with no regard for what is inside that is worth protecting, and without considering threats (intentional or unintentional) that are *already* inside, like malicious or foolish employees. This is only half the picture. In contrast, "keeping things in" by definition concerns itself with what is "inside" like sensitive data, privileged access, etc., and encourages the designer to consider *all* threats -- both internal and external -- against the protected resource.

Practically speaking, these two goals often result in different default policies. The goal of "keeping things out" often results in a policy that by default allows anything not considered to be a threat. This is called a **default allow** policy, and the classic example of this kind of firewall allows **all** outbound traffic, but only allows "untrusted" inbound traffic to special services, such as a web server (which is then responsible for its own security). This is better than nothing, but is hardly secure. If an attacker can trick someone inside into opening a [trojan horse](http://en.wikipedia.org/wiki/Trojan_horse_(computing)), the malicious software can exploit the liberal egress policy by making connections to a malicious host on the Internet, which can be used to send messages to the now-compromised system. Incidentally, this is how the firewalls on most home routers are designed.

On the other hand, the "keeping things in" policy usually results in a policy that by default *denies everything*, and allows only what is necessary for the proper functioning of a system. This embodies the principle of "[least privilege](http://en.wikipedia.org/wiki/Principle_of_least_privilege)" and in the context of a firewall is called a **default deny** policy. A firewall configured this way allows only the handful of things that are strictly required. This limits inbound traffic as before, but also only allows outbound traffic to carefully chosen targets. For example, this might only allow oubound traffic to a secured mail server, ssh server, and the few web servers required for an employee to accomplish their job. This drastically limits the means by which traffic can enter *or* leave the network, and if an employee executes a trojan as in the last example, that malicious software will not be able to contact its evil master because the malicious Internet host will almost certainly not be in the list of allowed outbound connections.

The obvious downside to a "default deny" firewall policy is maintenance and inconvenience -- it is harder to install in the first place, and any new network service or traffic type on the network must be explicitly allowed or it will not function. Allowing all outbound traffic significantly cuts down on this kind of maintenance -- at the cost of security.

### Firewall and Network Testing Tools

#### iptables: set and clear rules in netfilter

[iptables](http://en.wikipedia.org/wiki/Iptables) is actually the user space tool for administering the netfilter functions and tables in the Linux kernel, but the entire netfilter and iptables package is commonly referred to simply as iptables. iptables has several built-in tables of rules (such as filter and nat) , several built-in "chains" (which are sets of network traffic including the built-in INPUT, OUTPUT, and FORWARD for inbound, outbound, and routed traffic), a set of powerful loadable modules of matching stateful filters, the typical set of stateless criteria (such as source, destination, and interface), and a set of targets that represent what to do with a matching packet. These options allow sophisticated firewalls to be defined.

iptables can be intimidating and confusing at first glance even for veteran sysadmins, but especially to users who are not used to configuring firewalls at all or are used to configuring firewalls through a GUI. iptables expressive plugins further complicate the syntax. A typical iptables command looks something like this:

$ iptables -t filter -A INPUT -m state --state NEW -p tcp -s 192.168.0.1 --dport 23 -j REJECT

Upon closer inspection, iptables is revealed to be merely a command whose arguments define a single rule for packet filtering based on a number of possible criteria. iptables takes those arguments translates them one command at a time into priority-ordered filter rules in the Linux kernel. Thinking of iptables as a command with arguments can help demystify netfilter and the process of designing firewalls with iptables -- let's break down the above iptables command and translate it into English:

|  |  |
| --- | --- |
| **iptables command arguments** | |
| **command/argument** | **translation** |
| iptables | *We're going to use the iptables tool to insert a new rule into netfilter.* |
| -t filter | *This rule is going to go in the filter table, which is the built-in packet filtering table. This rule will apply only to:* |
| -A INPUT | *packets that have been put into the INPUT chain either by the kernel or by some previous rule and which:* |
| -m state --state NEW | *represent a new connection,* |
| -p tcp | *are Transmission Control Protocol (TCP) packets,* |
| -s 192.168.0.1 | *are from the host 192.168.0.1,* |
| --dport 23 | *and are destined for port 23.* |
| -j REJECT | *Reject any matching packet. Processing of all packets matching this rule will instantly jump to the built-in target REJECT, which means that the packet will be rejected by the kernel with some kind of network error message.* |

A few other examples:

$ iptables -p tcp --syn --dport 23 -m connlimit --connlimit-above 2 -j REJECT

This rule (from man iptables) allows 2 telnet connections per client host. Note that this rule uses the connlimit matching module, and rejects additional connections.

$ iptables -A INPUT -i lo -j ACCEPT $ iptables -A OUTPUT -o lo -j ACCEPT

These rules accepts any inbound or outbound traffic on the internal loopback network device (an internal, logical network adapter the kernel uses for network communication internal to the computer) regardless of state, protocol, source, or destination address. The -i lo and -o lo arguments specify the "input interface" and "output interface" the packet arrived on.

$IPTABLES -t filter -A INPUT -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT $IPTABLES -t filter -A OUTPUT -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT

These rules accept all INBOUND and OUTBOUND traffic regardless of interface, address, port or protocol. They use the state matching module, but accept all NEW, RELATED, and ESTABLISHED packets (which is basically all traffic). This rule is basically like having no firewall at all!

**NEW, RELATED, ESTABLISHED**

Think of your firewall as a security checkpoint in a big office building. There's usually two lines -- one for people with IDs, and one for people without IDs. If someone already has an ID, they can skip the long line, and go through. If they don't, they have to wait to get an ID card or visitor's pass. This is analogous to the distinction between **NEW** traffic versus **RELATED** or **ESTABLISHED** traffic (which you usually see together). Traffic marked **NEW** doesn't have an ID badge yet, because it is the first packet of a new stream of traffic. On the other hand, a packet of a **RELATED** or **ESTABLISHED** stream is part of something that by definition has already come through the firewall in the past. In other words, the firewall has already given that stream a "badge" (which is really an entry in an internal firewall data structure).

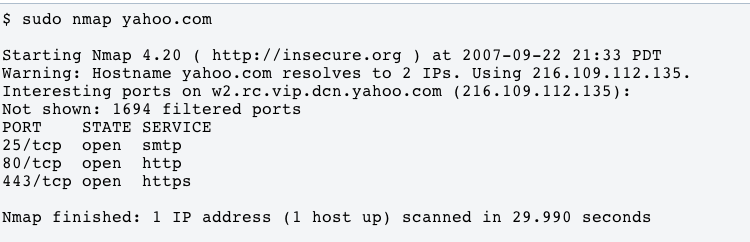
Among other things, this means that firewalls are typically structured so that the first section passes all accepted **RELATED,ESTABLISHED** traffic first, and then carefully allows only certain kinds of **NEW** traffic. Why do it in that order?

While this brief introduction to iptables should point you in the right direction, there are other features of iptables not included here that you may want to use for the exercise. There are many HOWTOs, tips, and tutorials online in addition to the iptables manpage; the exercise manual assumes that in order to complete the iptables exercise, you will need to [do some research](http://www.google.com) on your own.

#### nmap: network mapping port scanner

[Nmap](http://en.wikipedia.org/wiki/Nmap) ([homepage](http://www.insecure.org)) is a very popular "[port scanner](http://en.wikipedia.org/wiki/Port_scanner)" that can be used to determine what kind of services are running on a remote or local host, perform [OS fingerprinting](http://en.wikipedia.org/wiki/Passive_OS_Fingerprinting), and many other tasks. Nmap is capable of performing many tasks in a "stealth mode" designed to not raise the suspicion of the victim, but some tasks require more obvious techniques. We used nmap in ITEC 445.

Nmap is incredibly powerful, but the basic functionality of the application is easy to use:



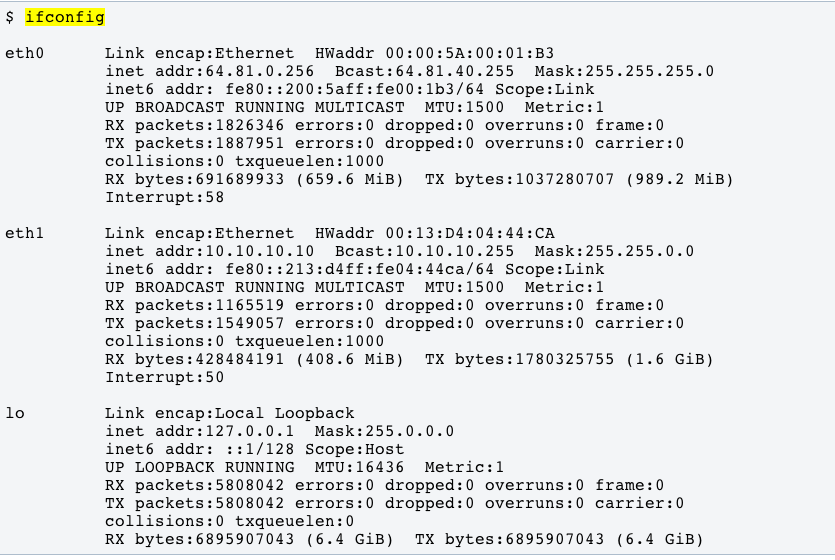
See the Nmap manpage or online documentation for advanced features.

#### ifconfig: configure Linux network devices

[ifconfig](http://en.wikipedia.org/wiki/Ifconfig) is the network interface configurator in Linux. It is most commonly used by users to see network addresses and statistics, but can also be used to enable and disable interfaces, set configuration options such as network addresses, and more.

For the purposes of this exercise, ifconfig will be used to determine what network addresses are running on what interfaces.

To see the current interface configurations:



Note that eth0's address is 64.81.0.256, while eth1's address is 10.10.10.10. This means that the two interfaces are on different networks.

#### telnet: cleartext remote shell

[TELNET](http://en.wikipedia.org/wiki/Telnet) (TELe-NETwork) is a cleartext remote terminal protocol. On its face, telnet is very simple; the user issues commands over a TCP socket, and the server replies with the results of those commands and waits for more input. In practice, this is complicated with various network and terminal emulation layers. Still, telnet is one of the simplest and oldest network protocols still in use. Due to its cleartext nature and low level access to the system, telnet is incredibly insecure -- it was common in the past for system administrators to log in as root using telnet on a hub network connection that could be sniffed by any sufficiently prepared attacker.

Thanks to the advent of Secure Shell (ssh), active use of telnet servers has died off except for some specialized uses. One place where telnet lives on is debugging ASCII-based network services. For example, web pages can be retrieved by telnetting to HTTP servers, and emails can be sent by telnetting to SMTP servers.

Telnetting to a suspected open port is still one of the fastest ways to see if a service is available or reachable.

Here are a few sample uses of telnet:



For this exercise, you will use telnet to test if a TCP port is open on a remote host. Telnetnetting to an IP and port (see above) should return a "connected" message if it is possible to connect to a running server.

#### netcat: a network swiss army knife

netcat (often nc on some systems) is a Unix utility for creating and using TCP and UDP sockets. In a very simplified way, netcat is like a telnet client and server without any built in protocol or terminal emulation. Another way of putting it is that netcat is the bare essentials for creating a TCP or UDP socket and client, with hooks for using standard in and standard out for IO.

There are too many cool uses of netcat to describe here. For the purposes of this exercise, we'll use netcat to create "fake" TCP or UDP servers that we can use to test firewall configurations.

##### Creating fake TCP/UDP servers with netcat

Starting a fake listening server (a program that will accept connections on a port) is as simple as running:

$ sudo nc -l 80 # we need sudo because 80 is a privileged port

... to start a listening TCP socket on port 80. Then, from the another host, you can either use telnet or nc to connect to the server you just started on the the first host. You should be able to type in one window and see output in the other if the network pipe is open.

Testing UDP services is exactly the same -- you can use netcat for that, too. You need to start a listening UDP process on the receiving side, and a sending process on the sending side. If you are testing UDP traffic from a client to a server, you can do something like this:

[server]$ nc -u -l 10000 # listen for UDP traffic on port 10000

Then on the client, do something like this:

[client]$ nc -u server 10000 # connect to server via UDP on port 10000

After establishing the connection, enter some data from standard input (probably your keyboard). Input on the sender should appear on the receiving terminal. Hit ^C to close the programs. UDP is of course an unreliable network protocol, so it's possible that there will be errors in the text file.

You can do any of this in reverse to test the connection from a server to a client. If you're able to transmit data, then the firewall is allowing communication.