Case Study: Setting up a Secure Network According to Open-Source Best Practices and Attacking it using Known Hacker Tools

Defensive Team: Max Planck, Ben Edwards, Tom Dawson
Offensive Team: James Garvin, Desh Gupta
Professor: Andrew Sung
Research Assistants: Srinivas Mukkamala, Guadalupe Janoski
CS-589: Topics on Information Warfare
New Mexico Tech University
Socorro, New Mexico

ABSTRACT:

This case study is intended to be used as a comprehensive how-to manual for setting up a resilient firewall utilizing best practices found across the open-source web. It also is intended to give a chronology of how a hacker team utilizing commonly available scripts might function. The defensive team utilized current open-source best practices in order to set up and protect a network running the same services as most small business organizations run today. The offensive team employed readily available script tools in order to attack the network. Useful resources, example commands, sources, results, and lessons learned are provided.

1. INTRODUCTION

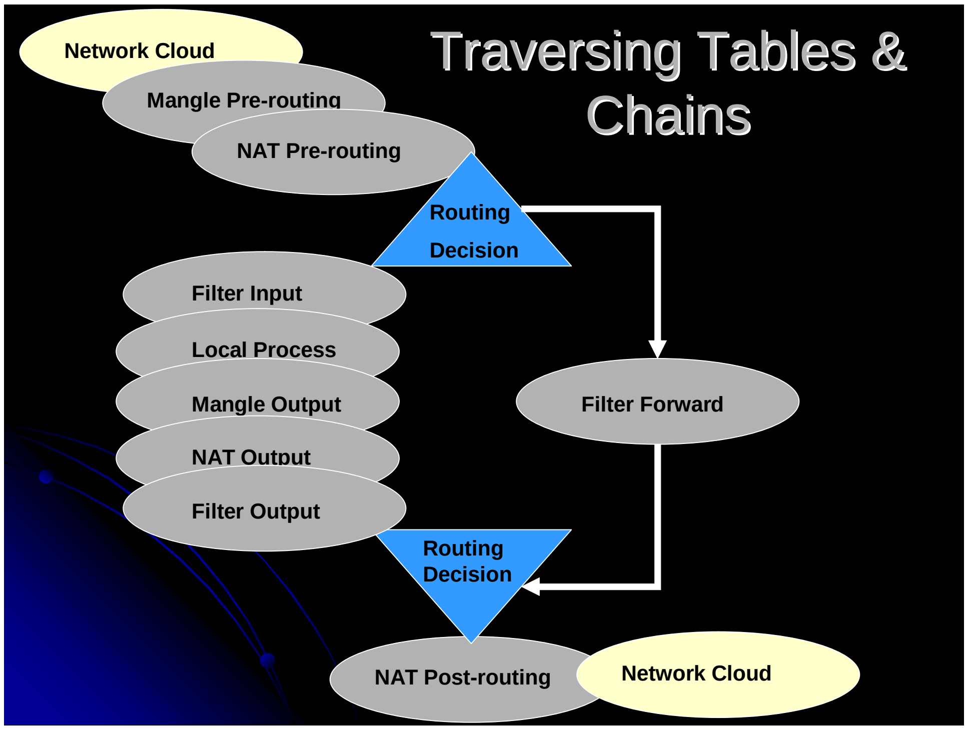
Many organizations default to using shrink-wrapped solutions for their fire-walls or outsource this function to managed security providers in order to minimize risk. This case study is intended to encourage more knowledgeable network administrators to use open-source code to construct firewalls that are extremely robust and resilient to attack. The firewall in this study was built with inexpensive hardware and free software. To implement this example in a larger organization would require increased hardware expenditures in order to optimize performance and minimize latency.

Another key benefit to configuring your own custom firewall is that it is not easily identifiable by attackers. If there are holes that are known to exist in any given firewall, then these will eventually find their way onto a hacking website. Specifically, the type of firewall we are creating does not announce itself with a custom banner, does not open ports with special services that are unique to a particular type of firewall (i.e. Checkpoint's Firewall-1 leaves port 257/tcp open for SNMP.) This custom firewall will also be resilient / misleading to fingerprint identification. Fingerprinting in this context refers to a machine's overall behavior to a set of packet stimuli.[1]

In general, the idea behind a state-ful firewall is simple. The processor on the firewall platform examines every incoming and outgoing packet in order to determine whether it fits into carefully determined rule-sets known as IPTABLES. If the packets meet pre-determined conditions, they are routed accordingly, if they do not, they are dropped. The following chart will

[1] http://www-net.cs.umass.edu/~brian/cs515/515-firewalls.ppt

walk through the steps that a packet moves through as it traverses the firewall's tables and chains.[2]



There are many ways of doing the various steps in this project. The methods that we chose are not necessarily the only ways of accomplishing this goal.

2. DEFENSE - SYSTEM DESCRIPTION AND RECOMMENDED BACKGROUND READING

A state-ful firewall was implemented on an Intel platform (500Mhz Celeron, 128 Mb RAM) running Redhat Linux Version 7.2 using iptables version 1.2.3 with the FTP hole patched. Behind this firewall a Sparc-10 (dual 50Mhz) running Solaris 8 was set up running internet services on port 80, e-mail services on port 25, and example data storage files located on the hard drive, etc.

We were provided with an entire subnet, with an IP address of 129.138.249.0/24. For security reasons we decided to use the minimum possible number of actual routable IP addresses. Also for heightened security, we setup the firewall to act as a router / Network Address Translator (NAT) for the machines behind it. This leaves only one routable/functional IP address from the initial subnet domain. If this configuration is going to be implemented for a larger company (or anyone with a large number of machines) then you may want to use multiple IP's from the address space in order to minimize traffic bottlenecking. For the purposed of this demonstration, the internal private IP space that we used was 192.168.1.0/24.

We have made every effort to make this guide self-explanatory, but before attempting to duplicate the commands in this paper, the network administrator should read over the

[2] http://iptables-tutorial.interans.com/iptables-tutorial/iptables-tutorial.html

hyperlinked sections out of the Red Hat Linux manual[3] and the Samba Netfilter guide.[4] In addition, we recommend reading over this TLDP site[5] and this Samba site[6] for networking and routing background information.

3. DEFENSE - STATEFUL FIREWALL CONSTRUCTION

First Step – Configuring the Linux box (firewall/router/NAT):

Below is the information that was changed from the default loading of the OS on the machine. The changes that we have made are located in the following directories and filenames. In order to re-implement this, some of the specific details (IP's, machine names, etc) will need to be tailored to suit your particular application.

Machine name of firewall/router/NAT: Alice

External IP:	129.138.249.121
Internal IP:	192.168.1.254

IP Information for Alice is established in:
/etc/sysconfig/network
/etc/sysconfig/network-scripts

External DNS for the machine is set in:
/etc/resolv.conf

Modules and dependencies:
/etc/modules.conf
/lib/modules/*"Insert kernel version here"*/ drivers/net

Location of the iptables file: /etc/rc.d/init.d/iptables

In order for the changes that you make to the iptables file to work, the previous rules need to be flushed from the system, and the new rules initiated. Redhat tends to treat this like a service, so the following line can be used to accomplish this:

```
service iptables restart
```

The default configuration starts by flushing and clearing all of the user defined iptables and ipchains rules so you are working from a clean slate. It then loads the rules that are laid out in the iptables file.

Please note that there is a difference between iptables and ipchains. Do not confuse the two of them. Even though their syntax is similar, they have different packet filtering semantics.

[3] http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/ref-guide/ch-iptables.html
[4] http://netfilter.samba.org/
[5] http://www.tldp.org/HOWTO/Adv-Routing-HOWTO.html
[6] http://netfilter.samba.org/unreliable-guides/networking-concepts-HOWTO/index.html

Second Step -- Setting Kernel Flags:

These lines may either be entered on the command line or inserted into the iptables file. For the sake of simplicity (and to help avoid typing errors) we recommend editing the iptables file.

If you are not intending for your box to forward traffic between interfaces, or if you only have a single interface, it would probably be a good idea to disable forwarding. Note that altering this value resets all configuration parameters to their default values, which are described by RFC1122 for hosts and RFC1812 for routers[1]. So, you'll want to modify this one before all other settings in /proc. This is done with the following line:

```
echo $"0" > /proc/sys/net/ipv4/ip_forward
```

This next line blocks Internet Control Message Protocol (ICMP) echo requests to broadcast addresses. This also prevents you from amplifying SMURF attacks. To disable responses to broadcasts, set the following kernel parameter "*icmp_echo_ignore_broadcasts*" to 1 by using:

```
echo $"1" > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

ICMP redirects can be used by malicious parties to effectively alter your routing tables, so we disable this as well. To disable the acceptance of redirected ICMP packets set the following kernel parameter "*accept_redirects*" to 0 by using:

```
echo $"0" > /proc/sys/net/ipv4/conf/all/accept_redirects
```

Normally, a host has no control over the route any particular packet takes beyond its first hop. It is up to the other hosts on the network to complete the delivery. IP Source Routing (SRR) is a method of specifying the exact path that a packet should take among the other hosts to get to its destination. Allowing IP source routing is generally a bad idea for the security conscious, as a malicious party could direct packets to you through a trusted interface and effectively bypass your security in some cases. A good example is traffic, such as SSH or telnet, that is blocked on one interface might arrive on another of your host's interfaces if source routing is used. This might not have been anticipated in your firewall settings. Attackers can compromise your network using source routing to generate traffic pretending to be from inside your network, but which is routed back along the path from which it came, namely outside. Source routing is rarely used for legitimate purposes. The following line disables source routing packets:

```
echo $"0" > /proc/sys/net/ipv4/conf/all/accept_source_route
```

Sometimes you will come across routers that send out invalid responses to broadcast frames. This is a violation of RFC 1122, "Requirements for Internet Hosts -- Communication Layers". As a result, these events are logged by the kernel. Some say that it is a good idea to turn this option off if you don't want to spend a lot of time deleting your fast-filling log files. However, if you want to be notified of what is going on (especially in the beginning) it is advisable to have this feature enabled. To enable this feature and protect your network against bad-error-messages, do the following:

```
echo $"1" > /proc/sys/net/ipv4/icmp_ignore_bogus_error_responses
```

Rp_filter performs the “add connection” feature and checks to see if libraries have changed, generates or reads in security cookies, and makes sure the user is authorized to make a given request. To enable spoof protection use the following:

```
echo $"1" > /proc/sys/net/ipv4/conf/all/rp_filter
```

Packets that have source addresses with no known route are referred to as "martians" within Linux. For example, if you have two different subnets plugged into the same hub, the routers on each end will see each other as “martians.” It is a good idea to log these packets in order to see what’s going on. In order to log spoofed, source routed, and redirected packets on your network, use the following:

```
echo $"1" > /proc/sys/net/ipv4/conf/all/log_martians
```

Third Step – Establishing the Rules:

We want to allow unlimited traffic on the loopback interface. The IP loopback interface is a unique interface in that it connects to itself. Packets sent to the loopback interface are simply reflected back to the IP stack, which then sees them as incoming packets. This interface is used mainly for testing and when services/daemons need to talk across the stack. To allow unlimited traffic on the loopback interface, use the following rules:

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

A common denial of service attack, known as SYN Flooding, is a method that essentially overwhelms a network interface by starting to establish a legitimate session, but never completes the handshake (or initiation protocol). Essentially, the network continues to hold open processes waiting for a complete handshake until it is eventually overwhelmed and truly legitimate users are unable to access the flooded network interface. There is no perfect solution to this type of attack. Our defense works by setting a maximum rate at which incoming connections can be established. After this connection rate is reached, the rest of the received SYN traffic is dropped. After the rate has settled down, SYN connections are allowed again. In this case study we have limited the connection set-up rate to three per second. Once that rate is achieved, additional connections are accepted at a rate of 1 per second (dropping all others, legitimate or otherwise) until the rate has decreased to below 3 per second. This may seem low, but in our case, this machine was not going to have a great deal of connections coming in from the outside world at any one given time during its legitimate usage. This defense is useful for the small organization, but is not practical for an internet-based business, as they need to respond to a wide variety of SYN traffic. You can of course raise the continuous and burst rates to suit your needs and desired traffic flows, but this is outside the scope of this project.

In the following commands we first create a user-defined table called “SYNflood.” We did this in order to prevent us from having to compare the non-SYN packets to all of the SYN rules, and vice-versa. It also allows us to be more efficient because we don’t have to add the rules twice (once in the input chain and once in the forward chain).

To create the table:

```
action "SYN FLood Protection:" touch /root/tmp
iptables -N SYNflood
```

To use the table:

```
 iptables -A INPUT -i eth0 -p tcp --syn -j SYNflood
 iptables -A FORWARD -i eth0 -p tcp --syn -j SYNflood
 iptables -A SYNflood -m limit --limit 1/s --limit-burst 3 -j RETURN
 iptables -A SYNflood -m limit --limit 1/min --limit-burst 3 -j LOG --log-prefix "Possible Syn
            Flood!"
 iptables -A SYNflood -j DROP
```

If any new requests for connections don’t start with a SYN, there is generally nefarious activity that is going on. We drop any such bogus traffic by ensuring that all new TCP connections must start with SYN packets:

```
action $"New Connections:" touch /root/tmp
iptables -A INPUT -i eth0 -p tcp ! --syn -m state --state NEW -j DROP
iptables -A FORWARD -i eth0 -p tcp ! --syn -m state --state NEW -j DROP
```

Fragments happen in normal traffic, but they are also a good indicator that nefarious activity is going on. For example, it was fragment flooding that Jolt2 used to take down Firewall-1 (a generally well respected piece of shrink-wrapped firewall software.)[2] In order to understand our network, we monitor these fragments in our log and then drop them. Different operating systems interpret fragments in different ways so dropping them is the safest course of action if you want a “stable” network. Many of these rules for fragments may seem redundant, because we also set flags in the kernel to drop fragments, but just in case there end up being bugs in the kernel later (or our IP stack gets corrupted somehow) it is better to be safe than sorry. Note that each rule is done once for the input chain and once for the forwarding chain. To deal with fragments in this manner do the following:

```
action $"Fragments: " touch /root/tmp
iptables -A INPUT -i eth0 -f -j LOG --log-prefix "Found a frag: "
iptables -A FORWARD -i eth0 -f -j LOG --log-prefix "Found a frag: "
iptables -A INPUT -i eth0 -f -j DROP
iptables -A FORWARD -i eth0 -f -j DROP
```

The following lines provide spoofing protection. This first line gives us a useful OK as it runs which lets us know where in the script we are currently executing. Please notice that we will get our OK regardless of what happens later on in the script.

```
  action "Spoofing: " touch /root/tmp # OK
  iptables -A INPUT -i eth0 -s 10.0.0.0/8 -j LOG --log-prefix "Spoofed A IP: "
  iptables -A INPUT -i eth0 -s 10.0.0.0/8 -j DROP
  iptables -A INPUT -i eth0 -s 192.168.0.0/16 -j LOG --log-prefix "Spoofed C IP: "
  iptables -A INPUT -i eth0 -s 192.168.0.0/16 -j DROP
```

```
iptables -A INPUT -i eth0 -s 172.16.0.0/12 -j LOG --log-prefix "Spoofed B IP: "
iptables -A INPUT -i eth0 -s 172.16.0.0/12 -j DROP
iptables -A INPUT -i eth0 -s 224.0.0.0/4 -j LOG --log-prefix "Spoofed D IP: "
iptables -A INPUT -i eth0 -s 224.0.0.0/4 -j DROP
iptables -A INPUT -i eth0 -s 240.0.0.0/5 -j LOG --log-prefix "Spoofed E IP: "
iptables -A INPUT -i eth0 -s 240.0.0.0/5 -j DROP
iptables -A FORWARD -i eth0 -s 10.0.0.0/8 -j LOG --log-prefix "Spoofed A IP: "
iptables -A FORWARD -i eth0 -s 10.0.0.0/8 -j DROP
iptables -A FORWARD -i eth0 -s 192.168.0.0/16 -j LOG --log-prefix "Spoofed C IP: "
iptables -A FORWARD -i eth0 -s 192.168.0.0/16 -j DROP
iptables -A FORWARD -i eth0 -s 172.16.0.0/12 -j LOG --log-prefix "Spoofed B IP: "
iptables -A FORWARD -i eth0 -s 172.16.0.0/12 -j DROP
iptables -A FORWARD -i eth0 -s 224.0.0.0/4 -j LOG --log-prefix "Spoofed D IP: "
iptables -A FORWARD -i eth0 -s 224.0.0.0/4 -j DROP
iptables -A FORWARD -i eth0 -s 240.0.0.0/5 -j LOG --log-prefix "Spoofed E IP: "
iptables -A FORWARD -i eth0 -s 240.0.0.0/5 -j DROP
```

Fourth Step – Explicitly Allow Desired Services into the firewall:

If we were going to have more services opened explicitly, then that would be done here in a similar fashion to what was used for opening Secure Shell (SSH) below. Port 22 is used for SSH. World Wide Web services would require ports 80 and 443 to be open. SMTP needs port 25 open. To allow incoming ssh sessions:

```
action "Allow Targets: " iptables -A INPUT -i eth0 -p tcp --dport 22:22 -j ACCEPT
```

Network Address Translation (NAT) is used to map one IP-space to another. In this implementation it has the added effect of hiding and relocating network services (that are located on a private network) from the outside network that it is connected to. This is extremely useful because it is nearly impossible to route a packet directly to any of the machines behind the firewall if they are invisible. For the same reason, it is impossible to route a packet with a private IP space source or destination across the Internet and have it be legitimate traffic. This essentially places all of the other addresses on the network “behind” the firewall. It does this by pre-routing our web requests that are coming into the firewall by telling them to go directly to the internal IP of 192.168.1.130 (on port 80).

To enable the NAT function:

```
action "NAT Table: " touch /root/tmp          # OK
```

```
# Direct all WWW packets to go through our web server by providing its internal address
# (this is known as Destination NATing or DNAT)
iptables -t nat -A PREROUTING -i eth0 -p TCP -d 129.138.249.121 --dport 80 -j DNAT –to
destination 192.168.1.130:80
```

```
# DNAT port 180 to our web server via the internal address and
# map it to port 22. This will allow external SSH access to the web box
```

```
iptables -t nat -A PREROUTING -i eth0 -p TCP -d 129.138.249.121 --dport 180 -j DNAT –to
destination 192.168.1.130:22
iptables -t nat -A PREROUTING -i eht0 -p UDP -d 129.138.249.121 --dport 180 -j DNAT --to-
destination 192.169.1.130:22

# Source NAT or SNAT everything going out of the internal network to the firewall’s external
#address
iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to-source 129.138.249.121

 # Accept things going to port 80 and 22 from the NAT.  This step explicitly allows these two
 #ports (22 and 80) through the forward chain.
 iptables -A FORWARD -p TCP -d 192.168.1.130 --dport 80 -j ACCEPT
 iptables -A FORWARD -p TCP -d 129.168.1.130 --dport 22 -j ACCEPT
```

Put simply, this ensures that the legitimate protocol traffic that is coming into the firewall in response to previous outgoing traffic is allowed back through the firewall. The state-ful nature of this firewall is provided by one of the core features of the Linux netfilter code, and should be well understood before being used in a protection environment. The administrator needs to understand why the states are matched in the way that they are.

```
# Create Stateful Monitoring
action "Stateful Rules:" touch /root/tmp        # OK

iptables -A INPUT -i eth0 -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -o eth0 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A FORWARD -i eth0 -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -o eth0 -m state --state NEW,ESTABLISHED -j ACCEPT
```

These final steps implement the primary philosophy of this firewall. This philosophy essentially denies everything and then opens up only the specific exceptions that are required for the services being run. Another common way to do this is to set the default policy of your tables to drop. Setting the policy on the tables << iptables –P *chain name* DROP>> rather than writing explicit rules to drop the unmatched traffic.

```
 # Drop all other traffic coming into the firewall
 action "Drop Rules:" touch /root/tmp           # OK
iptables -A INPUT -i eth0 -j DROP

# Drop all other traffic coming into the network
iptables -A FORWARD -i eth0 -j DROP
```

This is the last step to perform. This step prevents packets from accidentally going through the router before all of the firewall rules are in place.

```
# Enable ip forwarding
action "Start Routing: " touch /root/tmp        # OK
   echo $"1" > /proc/sys/net/ipv4/ip_forward
```

4. DEFENSE - SYSTEM MAINTENANCE AND MONITORING

Maintaining a firewall has two major components: the initial setting of the rules, and then the monitoring of those rules. A large percentage of the time administrators will create their firewall from whatever sources and then, once the firewall is up and running, they will forget about it. The effects of neglecting a firewall can be disastrous. A firewall needs to adapt as new attacks are created.

To this end the netfilter code in the 2.4 Linux kernel provides great logging facilities. These let the firewall give very verbose details of what its doing and of the traffic that its receiving. Between the logging of the netfilter code and the logging of the snort package, a detailed understanding of the traffic through a gateway can be obtained. However a lot of information is generated by this system. Some time should be taken to discover the balance between useful logging and logs that take to much time to look at. There are many open source packages that act as front ends to the logs of snort and netfilter, although they are not presented here, such packages make log analysis much easier and take less time.

The files of importance with the presented setup are
/var/log/messages
/var/log/snort/portscan
/var/log/snort
/var/log/secure

What to look for is somewhat of an art, however all systems eventually end up with a pattern in their logs which is generated by normal usage. Once this pattern is established it is generally easy to spot something out of the ordinary (this is where analysis tools are useful). Harmless attacks, like port scanning, are easy to spot because Snort tells you about them. DOS attacks are easy to spot in the logs but difficult to completely defend against. These lesser attacks can be signs of a larger attack to come. The most dangerous attacks are the root exploits and service attacks. Snort will tell you what it can about traffic to your open ports but it is only as good as its rules are up to date. This is also why an administrator should look at the logs generated by the individual services running on the system. Knowledge of the system is crucial to defending it.

Eventually, after you get a feel for the type of traffic your network is seeing, and how that is reflected in your logs, you will probably want to establish a few rate limiting rules into your logging statements. This will make them easier to go through (or make your PERL scripts run through them faster… however you prefer to do this).

Packet Sniffing: Installing SNORT[7]

Snort is a freeware packet sniffer that uses some of the advanced features of the libc6 library as an interface to the Linux Kernel tcp/ip stack code. Snort is able to analyze incoming packets for patterns. These patterns are determined by a set of rules that come with snort. The

[7] http://www.snort.org

ultimate goal of the pattern matching is to detect incoming attacks as they are occurring. This falls within the realm of intrusion detection, which is mostly outside the scope of this document. There are however, a few points that need to be made that are central to maintaining good security. These are: It is possible to make an attack within legitimate traffic, thus rendering your firewall useless, and in order to adapt to attacks you need to know what they are even if the firewall drops them.

Snort does not come with Redhat by default, so it must be downloaded and installed. Snort has documentation on how to do this on their website and within the downloaded package itself. Snort comes with a set of rules maintained on their website, these rules are excellent and up to date, and thus there is no real reason to modify them. If it is deemed necessary (i.e. you want to look for something special) the documentation that comes with the package explains how to add new rules. For installation help also see the snort how-to guide.[8] Once snort is installed, it will log what matches its rule base in a directory called /var/log/snort. In this directory snort will create sub-directories that are named based on the IP of an attacking machine. Within these sub-directories snort will log the actual packet that participated in the attack along with its time, source and destination. Snort will also put a note in the systemlog file (/var/log/messages) that something has occurred.

5. OFFENSE – A HACKER MONOLOGUE

The following section was written by the hackers themselves, using common jargon. In order to maintain the original feel of the writing, while still providing good educational definitions, we have inserted definitions we felt were required for the average semi-technical reader. Words with definitions have been highlighted, bolded and in the referenced footnote, you will find an active link to a website with a good definition of the word.

Hacking a system takes patience, diligence, and persistence. The focus of the hacking paradigm is to gather data, analyze the data, and attack. Each of these steps must be meticulously recorded. Data gathering includes targeting a system, packet sniffing, and probing. It is important to thoroughly record all data collected to ensure proper analysis. In the analysis phase, a hacker must also evaluate the security, response, and repercussions if the target is to be attacked. Finally, in the attack phase an attacker will attempt to circumvent the security of the target or attack the system in a specific manner. “Observe your enemies, for they first find out your faults” were words spoken by Atisthenes in 440 B.C., and are still true today. Within this section, words that might need a better definition are underlined, bolded and footnoted. By following the footnote hyperlink, the reader will find a full definition of the term.

First, we must acquire a target to gather data from. In this case the target is 129.138.249.254, and will subsequently be referred to as Target. We decided that on Target we would try to search for good information and possibly deface the website. To find out more about Target, a port scanner called nmap was used. Port scanning revealed very little about Target. Nmap produced this:

[8] www.linuxdoc.org/HOWTO/Snort-Statistics-HOWTO

```
Nmap (V. nmap) scan initiated 2.53 as: nmap -sS -sU -O -v -P0 -oN text.txt 129.138.249.121
Interesting ports on (129.138.249.121):(The 1540 ports scanned but not shown below are in
state: closed)
Port      State      Service
22/tcp    open       ssh
23/tcp    open       telnet
80/tcp    filtered   http
180/tcp   open       unknown
Sequence numbers: 2BBE5AC1 6C63AB5F B7156284 2B45594E B69FE13 E0138A1A
No OS matches for host (If you know what OS is running on it, see http://www.insecure.org/cgi-
bin/nmap-submit.cgi).
TCP/IP fingerprint:
TSeq(Class=TR)
T1(Resp=Y%DF=N%W=3FE0%ACK=S++%Flags=BAS%Ops=ME)
T2(Resp=N)
T3(Resp=Y%DF=N%W=3FE0%ACK=S++%Flags=ASF%Ops=ME)
T4(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=N)
PU(Resp=Y%DF=N%TOS=0%IPLEN=38%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULE
N=134%DAT=E)
```

Based on this first step, the firewall on Target appeared very restrictive. It managed to drop most packets. Usually a good way to gather data is to send short TTL (Time To Live) packets. These normally pass a firewall and help the attacker produce a rough topology of the network behind the firewall, especially if they have a valid checksum. In this case, the firewall squashed the short TTL packets.

Another problem was that the firewall dropped all non-legitimate ICMP, TCP, and UDP packets. This made efficient scanning almost impossible. Xmas tree and null scans were equally useless. The firewall, also known as “the Beast” was squashing almost everything we were sending at it. The firewall stopped us from doing an effective ping sweep. A ping sweep (also known as an ICMP sweep) is a basic **network scanning**[9] technique used to determine which of a range of **IP addresses[10]** map to live **hosts**[11] (computers). Whereas a single **ping**[12] will tell you whether one specified host computer exists on the network, a ping sweep consists of **ICMP[13]** (Internet Control Message Protocol) ECHO requests sent to multiple hosts. If a given address is live, it will return an ICMP ECHO reply. Ping sweeps are among the older and slower methods used to scan a network.

[9] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci802800,00.html
[10] http://searchwebmanagement.techtarget.com/sDefinition/0,,sid27_gci212381,00.html
[11] http://searchwebmanagement.techtarget.com/sDefinition/0,,sid27_gci212254,00.html
[12] http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci214297,00.html
[13] http://searchsystemsmanagement.techtarget.com/sDefinition/0,,sid20_gci214012,00.html

There are a number of tools that can be used to do a ping sweep, such as *fping*, *gping*, and *nmap* for **UNIX**[14] systems, and the *Pinger* software from Rhino9 and *Ping Sweep* from SolarWinds for Windows systems. Both Pinger and Ping Sweep send multiple **packets**[15] at the same time and allow the user to resolve host names and save output to a file.

To disable ping sweeps on a network, administrators can block ICMP ECHO requests from outside sources. However, many administrators fail to properly filter ICMP TIMESTAMP and **Address Mask Requests.**[16] Since all ICMP ECHO packets were dropped by this firewall, this potential vulnerability was not available for exploitation.

Even **sniffing**[17] was pointless. A packet sniffer can be considered as a sort of wire tap device. A device that can "plug" into computer networks and eavesdrop on the network traffic. Just as a telephone wiretap allows the CIA to listen to conversations, the same concept follows a packet sniffer in the sense that it allows someone to listen in on computer conversations.

Packet sniffers capture "binary" data passing through the network, most if not all decent sniffers "decode" this data into a human readable form. To make it even easier (for humans) another step occurs known as "protocol analysis". There is a varying degree of the analysis that takes place. Some are simple, just breaking down the "packet" information, while others are more complex giving "detailed" information about what it sees on the packet (i.e., highlights a password for a service).

One very important point to understand is that the sniffer has to be on the same "wire" on which the data is traveling to. In short the "probing" device that "captures" the data has to be on the same wire. The data can then be relayed to a decoding computer on a different network.[18]

We tried every type of scanning and probing we could think of. We even tried Firewalk. Firewalk is:

> " a technique that can be used to gather information about a remote network protected by a firewall. The purpose of the paper is to examine the risks that this technique represents. This paper is intended for a technical audience with an advanced understanding of network infrastructure and TCP/IP packet structures.
>
> Firewalking uses a traceroute-like IP packet analysis to determine whether or not a particular packet can pass from the attacker's host to a destination host through a packet-filtering device. This technique can be used to map 'open' or 'pass through' ports on a gateway. More over, it can determine whether packets with various control information can pass through a given gateway. Also, using this technique, an attacker can map routers behind a packet-filtering device. To fully understand how this technique works, we first

[14] http://searchsolaris.techtarget.com/sDefinition/0,,sid12_gci213253,00.html
[15] http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212736,00.html
[16] http://support.microsoft.com/default.aspx?scid=kb;EN-US;q170292
[17] http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci802721,00.html
[18] http://www.surasoft.com/tut/packsniffing.htm

need to understand how traceroute works. This paper provides an introduction to traceroute." [19]

In short, Firewalking produced nothing. A similar tool called Cheops was employed, but it wasn't useful. We used nmap, short ttl packets, simple ping sweeps, packet sniffing, and firewalking. We ended our scanning with very little data, and a ton of frustration.

With the little data we did gather we needed to analyze what we had. We found that port 25/tcp was open. This meant that a telnet attack was possible. It was good news, but we weren't sure it was feasible to do such an attack with such a strict firewall. Notice that port 80/tcp is also open, but filtered. This means that one of our initial objectives was possible. We could deface the website. However, after a little more data collection, we found that the web server was Apache. This is bad news because we couldn't find any posted security-hole-reports for Apache.[20] We probed a little deeper to see exactly how secure the system was. We decided to first attempt a Reverse Telnet attack. This is a good way to get in if X Windows isn't on the machine. X Windows is the graphical user interface for the Linux and Unix families. "It provides a public protocol by which client programs can query and update information on X servers." [21]

The telnet connection will originate from the system to which the attackers are attempting to gain access. Telnet is usually not restricted, so this is easy to do. Netcat (NC) will help us to attack. NC "is a simple Unix utility which reads and writes data across network connections, using TCP or UDP protocol. It is designed to be a reliable backend tool that can be used directly or easily driven by other programs, it is a feature-rich network debugging and exploration tool, since it can create almost any kind of connection you would need and has several interesting built-in capabilities."[22]

After several steps a back channel is created and we have a shell on the target. It seems the firewall was setup in such a way that even getting to the machine running Apache was impossible.

We also tried to analyze how the firewall dropped packets. We tried a couple of different SYN **flood**[23] attacks, a TCP flood, a ping flood, and a couple of slightly different UDP floods. It seems that the firewall is pretty efficient at dropping packets and flooding seems to be a pointless endeavor. The floods used create a stream of requests that overwhelms a server and can cause a denial of service to the legitimate users.

We worked on trying to fingerprint the OS as well. The firewall wouldn't allow direct OS fingerprinting. So I took a guess as to what it was running. I guessed that Debian Linux was the OS that was running the Apache web server. So I focused on attacking **Debian**[24] which was

[19] http://www.packetfactory.net/Projects/Firewalk/firewalk-final.html#_Toc433604122
[20] http://httpd.apache.org/docs/misc/security_tips.html
[21] http://www.cbbrowne.com/info/x.html
[22] http://freshmeat.net/projects/netcat/?topic_id=150
[23] http://www.tuxedo.org/~esr/jargon/jargon.html#flood
[24] http://www.debian.org/ and http://www.linuxsecurity.com/

incorrect. We didn't get a chance to attack the website. However, some slightly more advanced attacks were performed by William Colburn. He tried three methods of attack. The SSH root exploit, port walking, and some advanced scanning techniques were used. The SSH root exploit[25] was tried, but to no avail. Port walking, one of the more interesting attacks, was used to try to deny service from the system. Port walking opens connections which causes the target to open ports to reply. These ports open one after another and take system resources on the targets TCP stack. The slightly more advanced port scanning such as XMAS tree and null scanning were used. These are more clandestine methods of scanning so most firewalls and packet filters let them slip through. XMAS uses FIN, URG, and PUSH flags on the packet and null scan turns all flags off. Generally all the attacks failed, but much was learned after each attack. Had we more time it is possible that we could have produced some results.

6. OFFENSE – LESSONS LEARNED

We attacked an extremely secure system that focused on keeping attackers out. The firewall was very strict, thus causing little chance to flood, short TTL through, or attack with reverse telnet. However, since there wasn't much to attack that limited our ability and well as the lack of social engineering involved with the project. In hind sight it would have been nice to have a larger amount of hack-able items as well as the inclusion of "real world" social engineering. Generally, the attacks went well even though they failed and the information gained from attacking such a secure system was invaluable.

The firewall was the strictest firewall I have even encountered. It seems to either drop everything or filter it to such a point that it destroyed viable attacks. The firewall could quickly squash a flood of any packet type and the worst that could be done to the system would be to slow it down for a time. This isn't exactly a denial of service (DoS) attack, but it is causing problems. A major trick for hackers is using short TTL packets to trick a firewall. Most firewalls need to accept short TTL packets, or at least filter them somewhat. This firewall stepped on all short TTL packets that were sent. While this is secure, it is dubious whether this would work in the real world. The final item in most hackers bag of tricks is reverse telnet. While this is an interesting attack, it isn't always viable. Since I misidentified the system that I was to reverse telnet, the attack was unsuccessful. The most interesting part of this project was gathering data. Most of the time spent on this project was looking for security holes, back doors, or exploits. Hacking is mostly research, some coding, and a few minutes for each attack. Most of these attacks are hard to implement in practice and thus taught us quite a bit about gathering information as well as developing viable attacks given that data.

The largest problem was the lack of viable targets. It seems the targets were limited and didn't totally reproduce a "real world" type system. It would have been more interesting if the network behind the firewall was a small representation of a typical business. Thus, multiple operating systems, more viable targets, and a slightly less strict firewall would have produced slightly more realistic results. Social engineering would have added another dimension to the project. While it is mostly ignored, social engineering can produce results when technology fails. Generally, users aren't trained well or they are easily persuaded. The "Jedi mind trick"

[25] http://www.ssh.com/products/ssh/exploit.cfm

can be employed to gather sensitive information and typically not alert security to your presence. Perhaps if this project is repeated this could be added.

The red team attacks went well and we were able to attack a system without fear of repercussions. The attacks used were mostly limited to simple attacks, but this was due to the strictness of the firewall. It seems the worst that could be done to the system is to slow it down, not even a true denial of service. The project improved my research ability and lexicon of bugs present in the system. The project was a wonderful learning experience, but it could be added to if done again.

7. DEFENSE – LESSONS LEARNED

It is with some satisfaction that the defensive team looks back on this episode. The firewall was not our original creation, but it was a compilation of best practices that we have gleaned from years of work and practice with Linux. Although one is generally more confident because of anonymity when hiding behind a non-shrink-wrapped firewall, it is very good to put your system to the test, to learn what attacks have validity, and to try and come up with innovative fixes. In retrospect, the most valuable attribute we employed was diligence in implementing patches, and the persistence to scour the web in search of known problems and solutions.

It is true that if this firewall were implemented in a live business there would be more vulnerabilities, but those vulnerabilities can be minimized by consistent user education combined with disciplined vulnerability repairing and log monitoring.

8. ADDITIONAL VALUABLE RESOURCES

Example scripts and tutorials:

http://www.boingworld.com/workshops/linux/iptables-tutorial/iptables-tutorial.html#RCFIREWALLFILE

http://www.linuxguruz.org/iptables/howto/maniptables.html