|  |  |
|---|---|
| 1. Semester and Year | : 2012 FALL |
| 2. Course Number | : CS-227 |
| 3. Course Title | : Linux & System Software |
| 4. Work Number | : LA-10 |
| 5. Work Name | : Lab 10 |
| 6. Work Version | : Version1 |
| 7. Long Date | : Sunday, 4, November, 2012 |
| 8. Author(s) Name(s) | : Jake Waffle |

**Task 7.1**

For this task I will be learning about network security.

**Shutting Down Unused Servers**

When it comes to security, it's good to keep in mind that unused servers are able to be exploited  and used for malicious goals (Smith, 300.) So it's good practice to make sure that unused servers are shutdown. Then people won't be able to try any exploits on your unused server.

To be able to shutdown unused servers, you first need to identify the servers that are active (Smith, 300.) This can be done with the netstat command along with the -l and -p options.

```
#netstat -lp
```

The -l option will provide a list of the servers that are waiting for an external connection (Smith, 300.) While the -p options will display the name and process ID that of the process that's listening for connections. But when it comes to servers that are being handled by a super server, the PID will refer to the super server. So referral to the super server's configurations are necessary in order to determine the server that is being handled by the super server.

After identifying the active servers, we can then decide whether or not the server is needed or not. And with the list of unwanted servers, we can then go about shutting them down (Smith, 301.) This can be done simply by uninstalling the server. Or if that isn't possible one may be able to remove the SysV startup script for the server in the runlevel's configuration. And there is also the method of  removing the configurations from the super server (for the servers that are being managed by a super server I assume.)

## Using Server Access Rules

Depending on the server, it may have a unique set of access control methods available to the user (Smith, 301.) Mr. Smith gives an overview for the common control methods that are usually available to a user on many servers (it seems that allow/deny are good keywords to look for.) These methods are usually used to allow or deny an IP address or a hostname from being able to talk with the server. And Smith states that is usually best to use IP addresses instead of hostnames for the allow/deny methods.

## Using xinetd Access Restrictions

First off, xinetd is said to be a super server that is common amongst Linux distos (Smith, 301.) A super server manages a number of other servers individually. Each server is handled by its own file within the /etc/xinetd.d directory. And within these files or the default section in /etc/xinetd.conf several options can be used to set up access restrictions.

These options are very much similar to our previous section's (Using Server Access Rules) common control methods.

There is the only_from (allow) option that allows specific IP addresses or hostnames to access the server (Smith, 302.)

```
#only_from 192.168.78.0
```

There is also the no_access (deny) option that denies IP addresses or hostnames from accessing the server (Smith, 302.)

```
#no_access 192.168.78.0
```

Finally there is the access_times option that sets a specific range of time for the server to allow access (Smith, 302.) So outside of the range, no IP addresses or hostnames will be able to access the server. The following example allows access only from 10:00AM to 4:30PM.

```
#access_times 10:00-16:30
```

**Using TCP Wrappers**

Mr. Smith in the book says that it's possible to use the inetd super server in conjunction with a package, called TCP Wrappers, in order to give inetd the same sort of access controls as xinetd (Smith, 302.) It seems that the inetd super server calls tcpd when a client tries to access a server, instead of talking to the server directly. Then tcpd determines if the server is accessible to the given client and then calls the server if the client is accepted.

/etc/hosts.allow and /etc/hosts.deny are the two files that are consulted when determining if a client is alright to access a server – although Smith says they are actually one file (Smith, 303.) There uses seem very similar to the last two sections alright. Within the /etc/hosts.allow file, one can add in IP addresses or hostnames that are allowed to access one or many servers handled by inetd. And similarly within the /etc/hosts.deny file, one can add in IP addresses or hostnames that aren't allowd access to one or many servers handled by inetd.

The format for both of the /etc/hosts files are identical (Smith, 303.)

```
#Daemon-list : client-list
```

Daemon-list is the list of servers that are allowed/denied by the specific client-list of IP addresses and hostnames. The wildcard “ALL” is able to be used for both all servers (daemon-list) and all clients (client-list.) But the “EXCEPT” keyword can only be used in the client-list. The “EXCEPT” keyword only seems useful when flagging an exception for a network that denotes the client-list. And a network can be denoted by a leading dot (.domain) for the domain and a trailing dot (numbers.) for IP address blocks.

**Setting Up Local Firewall Rules**

A Linux kernel is capable of filtering packets based on low-level attributes: source/destination Ip addresses, source/destination port numbers and whether or not the packet was from an existing session (Smith, 303.) In order to do this, kernels have packet filtering tools that allow the altering of a kernel's packet filter table. These packet filtering tools are backwards compatible, so older ones will work on newer kernels. But the newer packet filtering tools will utilize more features on the newer kernels.

The kernel's packet filter table defines a default policy for its packets that don't follow any of the set rules (Smith, 304.) The default for the default policy is ACCEPT and that accepts packets that don't follow our given rules. But to add to the security, we can change this to DROP or REJECT.

To actually create the rules for the firewall, we can use a package filtering tool, called iptables (Smith, 304.) Using this command we can define a set of rules for each chain of the kernel's filter table (with the -A option.) It takes multiple rules to make up a complete chain. And before we update the chains with the rules that we want, we have to flush it (with the -F option) so that it is updated within the kernel's filter table. We can also change a chain's default policy (with the -P option.)

The book then goes into the handling of expected traffic through iptables. Loopback traffic, DNS traffic, Client traffic and SSH server traffic are the different kinds of traffic listed in the book (Smith, 307.) But server support is left out and may need accounting for if they are used.

Loopback traffic to and from the loopback interface (127.0.0.1) is expected to be accessible by certain Linux tools (Smith, 307.) And since it's local, there isn't too much risk in accepting its traffic. But however it's possible for one to spoof someone's computer using the loopback interface's address. I'm not entirely sure on how the book uses the lo interface to prevent that though

DNS traffic can be handled by enabling UDP traffic to and from port 53 (Smith, 307.) The local DNS server is necessary on most systems when looking up hostnames or addresses. It's possible to strengthen this configuration by specifying the DNS server's address when setting the rules. Although multiple DNS servers will require individual rules for each one.

Client traffic can be handled by, “enabling TCP packets to be sent from unprivileged ports (those used by client porgrams) to any system” (Smith, 307.) These unprivileged ports are also setup so that servers are unable to run on them. This prevents intruders from logging into unauthorized servers.

SSH server traffic can be handled by enabling TCP packets to be sent to and from port 22 (the SSH server port)(Smith, 307.) This access is also restricted to the computer's local network (172.24.1.0/24.)

**Setting Up a Stand-Alone Firewall**

In the book's example script they basically are just blocking all INPUT for the router's network (192.168.24.0/24) along with outside access to some key ports (Smith, 308.) The outside INPUT to the router's network are just dropped. And the udp/tcp packets being sent (via FORWARD chain) to the samba and SMTP ports are also dropped. And of course the three chains are flushed at the beginning and had their policies set.

**Script Task:**

Write a script that logs off of all users, updates the system, reboots machine and set it up to run at 02:00AM every day.

```
#script.txt
#!/bin/bash
echo “Shutting down the system for updates in 30 mins!”
sleep 1800
killall -u
sudo apt-get update
sudo apt-get upgrade
reboot
```

The above script simple does most of the tasks on the above task. The killall command kills all processes, but with the -u option it will only kill (log off) user processes. The apt-get commands are used for updating the package list and upgrading your software. Then the reboot command reboots the system.

```
0 2 * * * ./script
```

Finally the above line if placed in a crontab will execute the above script every day at 02:00AM. But keep in mind that the script has to be in one of the directories in the $PATH variable for the crontab to be able to find it.

## References

-Roderick W. Smith. linux administrator StreetSmarts. Induanapolis, Indiana: Wiley Publishing, Inc, 2007