Interested in learning more about security?

SANS Institute InfoSec Reading Room

Using Linux Scripts to Monitor Security

This paper will show how to use basic Linux scripting to create a reusable network security monitor that is easy to use and easy to maintain. The purpose of this exercise is introduced with suggestions where it might be useful. Linux commands are discussed, along with techniques to automate them and interpret their results. Methods for turning these scripts into a generic, reusable tool that is easy to maintain are demonstrated, along with further suggestions for enhancing this tool. Various examples are given to show ...

Using Linux Scripts to Monitor Security

by
Harvey Newstrom

GSEC Practical Requirements (v.1.4)
Option 1 – Research on Topics in Information Security
August 23, 2002

Table of Contents

1. Abstract

This paper will show how to use basic Linux scripting to create a reusable network security monitor that is easy to use and easy to maintain. The purpose of this exercise is introduced with suggestions where it might be useful. Linux commands are discussed, along with techniques to automate them and interpret their results. Methods for turning these scripts into a generic, reusable tool that is easy to maintain are demonstrated, along with further suggestions for enhancing this tool. Various examples are given to show how these techniques can be applied to various security requirements. The full script including all the examples and the complete output are given at the end of the paper, along with a list of references. This should be enough information for security professionals to start creating their own generic reusable Linux scripts within their own collection of personal tools.

2. Purpose

Many security professionals find themselves in a position where they do not have a complete set of commercial security tools. This can occur at a client site where the tools are lacking, in a low-budget situation where tools cannot be purchased, or in limited environments where commercial products cannot be implemented. Even in limited situations, it is unacceptable to try to implement security without proper tools. Even where extensive commercial tools are unavailable, each security requirement must be addressed in some way. Where extensive tools are not available, simpler tools of some sort must be implemented to enable basic security.

This is where Linux scripting can help. Linux has many powerful commands that can be utilized to check various aspects of security. These commands can be harnessed and automated into a generic reusable tool that will provide the desired results. While such scripts may not always rival extensive commercial products, they do provide basic functionality, consistency, ease of use, and ease of maintenance.

This paper will demonstrate how to create such a generic tool. The examples are written in the bash (GNU Bourne-Again shell) scripting language, since this is a default shell that is available under most Linux systems. Any other shell or scripting language can be used as well. Perl provides an excellent scripting language, but may not be implemented in all environments.

3. Linux Commands

Paul Sheer in "Linux: Rute User's Tutorial and Exposition" (http://rute.sourceforge.net/node51.html) says that "Unix systems are the backbone of the Internet. Heavy industry, mission-critical applications, and universities have always used Unix systems." There are many powerful Linux commands that can be used to check various aspects of network security.

The *ping* command will show whether a machine is available on the network or not. It can also show timing delays and whether packets are being lost on the network.

The *arp* command will lookup the MAC (media access card) hardware address of a machine. This can be used to verify that the IP address being pinged is actually the correct hardware device to be checked and not some other device that has been misconfigured with the same IP address.

The *traceroute* command will show the route taken by packets to a specific IP address, as well as the IP address of each router between the two. It also will show timing delays and whether packets are being lost on the network.

The *netstat* command will show what connections are currently active between the local machine and other network machines. On a server this would show who is connected to the server or communicating with it.

The *cksum* command or *md5sum* command will produce a checksum for files or textual output from another command. This can be used to determine if these have changed since previous checks.

The *lynx* command will dump content from web pages. This can be used to access web-enabled devices on the network and to obtain data from them.

The *snmpwalk* command will show various MIB (management information base) variables from snmp-compliant network devices. This can be used to query such devices and obtain a lot of detailed statistical and status information on their operations.

The *nmap* command is a network exploration tool and security scanner. This can be used to scan IP address ranges to see if any unexpected machines are found. It also can be used to scan specific ports on specific machines.

The *netcat* command will scan a range of ports on a specified machine and report the response from each. For many ports, this will even report the software and version number running on the port. This can be used to check software version levels of machines on a network.

4. Automating Linux Commands

Basic scripts can be created to automate these commands. This allows a single script to perform frequent tests. The script can contain hard-coded commands so that command syntax does not have to be remembered and so helpful options are not forgotten. Whenever a new technique is discovered, it can be added to the script for future use. The first chapter of the Advanced Bash-Scripting Guide (http://mirrors.sunsite.dk/ldp/LDP/abs/html/why-shell.html) by Mendel Cooper states that "A shell script is a 'quick and dirty' method of prototyping a complex application" and that "Shell scripting hearkens back to the classical Unix philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities."

Consider this simple environment consisting of a server, five PCs and a DLS router connecting to the Internet:

```
Machine:      IP Address:
router        192.168.1.1
pc1           192.168.1.201
pc2           192.168.1.202
pc3           192.168.1.203
pc4           192.168.1.204
pc5           192.168.1.205
server        192.168.1.254
```

These devices can be tested by interactively typing the *ping* command with a count of three and observing the results:

```
➢  ping -c3 router

PING router (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=150 time=0.451 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=0.428 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=150 time=0.428 ms
--- router ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.428/0.435/0.451 ms
```

This shows that the router machine is up on the network. It shows that it is not losing packets and that its round-trip times appear good. However, it would be time-consuming to frequently ping every machine and observe these results. A better method would be to automate a script to automate these commands. Instead of repeating the above commands for each machine, consider the following script called "report":

```
#!/bin/bash
# "report" script to report network status
ping -c3 router
ping -c3 pc1
ping -c3 pc2
ping -c3 pc3
ping -c3 pc4
ping -c3 pc5
ping -c3 server
```

This script can be typed into a file using any text editor, such as vi or emacs. The first line tells the Linux system which shell to use to execute this script. (This paper uses the /bin/bash shell because it is a common default on most Linux systems.) The second line is a comment line starting with "#" and space. The subsequent lines are the commands that the script will execute. This script will execute the seven *ping* commands for when the command name of the script is typed at the Linux prompt:

```
➢ report

PING router (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=150 time=0.454 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=150 time=0.425 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=150 time=0.420 ms
--- router ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.420/0.433/0.454 ms
PING pc1 (192.168.1.201): 56 data bytes
--- pc1 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
PING pc2 (192.168.1.202): 56 data bytes
64 bytes from 192.168.1.202: icmp_seq=0 ttl=255 time=0.383 ms
64 bytes from 192.168.1.202: icmp_seq=1 ttl=255 time=0.360 ms
64 bytes from 192.168.1.202: icmp_seq=2 ttl=255 time=0.371 ms
--- pc2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.360/0.371/0.383 ms
PING pc3 (192.168.1.203): 56 data bytes
64 bytes from 192.168.1.203: icmp_seq=0 ttl=255 time=0.357 ms
64 bytes from 192.168.1.203: icmp_seq=1 ttl=255 time=0.323 ms
64 bytes from 192.168.1.203: icmp_seq=2 ttl=255 time=0.317 ms
--- pc3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.317/0.332/0.357 ms
PING pc4 (192.168.1.204): 56 data bytes
64 bytes from 192.168.1.204: icmp_seq=0 ttl=64 time=0.725 ms
64 bytes from 192.168.1.204: icmp_seq=1 ttl=64 time=0.272 ms
64 bytes from 192.168.1.204: icmp_seq=2 ttl=64 time=0.236 ms
--- pc4 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.236/0.411/0.725 ms
PING pc5 (192.168.1.205): 56 data bytes
64 bytes from 192.168.1.205: icmp_seq=0 ttl=255 time=0.359 ms
64 bytes from 192.168.1.205: icmp_seq=1 ttl=255 time=0.341 ms
64 bytes from 192.168.1.205: icmp_seq=2 ttl=255 time=0.324 ms
--- pc5 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.324/0.341/0.359 ms
PING server (192.168.1.254): 56 data bytes
64 bytes from 192.168.1.254: icmp_seq=0 ttl=255 time=0.108 ms
64 bytes from 192.168.1.254: icmp_seq=1 ttl=255 time=0.070 ms
64 bytes from 192.168.1.254: icmp_seq=2 ttl=255 time=0.072 ms
--- server ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.070/0.083/0.108 ms
```

5. Interpreting Results

While the *report* command certainly is easier to type than all those *ping* commands, the output is not any easier to read. To simplify the output, there needs to be added functionality in the script. Instead of just automating the commands, the script should capture the output of these commands, and

automate the evaluation of their results. Consider the following more advanced version of the report script:

```
#!/bin/bash
# "report" script to report network status

ping=`ping -c3 router | tail -2`
loss=`echo $ping | cut -d"," -f3 | cut -d" " -f2`
delay=`echo $ping | cut -d"=" -f2 | cut -d"." -f1`

if [ "$loss" = "100%" ] ; then
   echo router is not responding at all
elif [ "$loss" != "0%" ] ; then
   echo router is responding with packet loss
else
   if [ "$delay" -lt 100 ] ; then
      echo router is responding normally
   else
      echo router is responding slow
   fi
fi
```

This script not only pings the router machine, but also interprets the results to report if it is losing packets or responding slowly.

The first executable line of the script (ping=`ping –c3 router | tail –2`) captures the desired *ping* results and stores it in a variable named "ping". The "ping=" sets the variable named $ping to whatever results from the following commands inside the reverse single quotes (`…`). The first command inside the reverse single quotes (ping –c3 router) pings the router machine and produces its normal output. The pipe character (|) forces the output to go through the following command instead of to the screen as usual. In this line, the following command (tail –2) extracts the last two lines of the output. It is these last two lines that get stored into the $ping variable for later use. These two lines contain the packet-loss report and the round-trip report.

The second executable line of the script (loss=`echo $ping | cut -d"," -f3 | cut -d" " -f2`) extracts the percent of packet-loss from the *ping* results and stores it in a variable called "loss". The "loss=" sets the variable named "loss" to whatever results from the following commands inside the reverse single quotes (`…`). The first command inside the reverse single quotes (echo $ping) merely outputs the contents of the $ping variable which contains the *ping* results. The dollar-sign ($) in the beginning of the variable name references the contents stored in this variable and not the literal word "ping" itself. The pipe character (|) forces the output to go through the following command. In this case, the following command (cut -d"," -f3) extracts the third field of text delimited by commas (,). This is then piped with another pipe character (|) into another command (cut -d" " -f2) which extracts the second field delimited by spaces. The end result of this is to cut off the output after the second comma and then cut out output between the first and second spaces. If the *ping* output produces "3 packets transmitted, 3 packets received, 0% packet loss" the script would capture the "0%" in the $loss variable.

The third executable line of the script (delay=`echo $ping | cut -d"=" -f2 | cut -d"." –f1`) extracts the best delay time from the ping results and stores it in a variable called "delay". It is similar to the above line, except in this case it is extracting the number after the equals-sign (=) and before the decimal point (.) for the value. This is the number of whole milliseconds that the round-trip ping packet took to return. If the ping output produces "round-trip min/avg/max = 0.423/0.436/0.451 ms" the script would capture "0" in the $delay variable.

The next paragraph of the script tests these stored variables to interpret the results and to output conclusions based on these results. The first *if-then* statement (if ["$loss" = "100%"] ; then) decides whether the loss is 100%. The brackets ([…]) invoke the test program to evaluate whatever expression is in-between them. In this case, it tests whether the variable $loss is equal to (=) "100%". The semi-colon (;) allows the *"if"* and *"then"* commands to be put on the same line. This *if-then* sequence will execute the following command if the above test is true. If loss is 100%, the following command (echo router is not responding at all) will report. Only if the above test is false will it continue on with the *"elif"* command. This command (short for "else if") specifies an alternate test to try if the previous one fails. If loss wasn't 100%, it will then test to see if it was not "0%" (elif ["$loss" != "0%"] ; then). If so, it will report it (echo router machine is responding with packet loss). Otherwise, it continues to the default (else) case. If all else fails, the *if-then-else* series will execute this last case. The script has already tested for total loss and partial loss. The remaining case is no loss. In this case, it tests for slowness instead. If $delay is less than (-lt) 100 milliseconds, it reports that router is responding normally, otherwise it reports that router is responding slowly. (The exact timing that should be considered "slow" will vary for different environments.)

The above script will perform the *ping* commands, test the results, and give the final conclusions in a simplified form. The commands do not have to be executed separately, the results read, or conclusions drawn. The usage and output of this script would be simple:

```
➢ report
router is responding normally
```

Note that the simplification of the output has complicated the script. The complexity of command interpretation has been moved from the user into the script itself. It performs this work for the user. However, an unpleasant side effect of this is that the script can become very unwieldy. The above script only tests a single machine. It would have to repeat those lines for all the other machines. This would produce a very long script that is very difficult to maintain.

6. Creating a Generic Tool

Steve Parker says in the Bourne Shell Programming section of his Shell Scripting Tutorial (http://steve-parker.org/sh/philosophy.shtml) that "A clear layout makes the difference between a shell script appearing as 'black magic' and one which is easily maintained and understood." It is not sufficient to merely simplify the invocation and results of the script. The script itself must be simple to maintain. If the script itself becomes too complex, it merely replaces network maintenance tasks with script maintenance tasks. It becomes difficult to add or remove devices to check, and to keep the script working without errors.

To solve the scripting complexity problem, this complexity can be separated from the parts of the script that need to be maintained. This can be done by moving the complex portions of the script out of the way into subroutines. Once a subroutine is working, it does not need to be maintained or changed as the script is modified. Instead, there will be a much simpler list of subroutine calls that will be edited. Consider the script after it is enhanced to use a subroutine to perform the ping tests:

```
#!/bin/bash
# "report" script to report network status

pingtest() {
   ping=`ping -c3 $1 | tail -2`
   loss=`echo $ping | cut -d"," -f3 | cut -d" " -f2`
   delay=`echo $ping | cut -d"=" -f2 | cut -d"." -f1`

if [ "$loss" = "100%" ] ; then
   echo $1 is not responding at all
elif [ "$loss" != "0%" ] ; then
   echo $1 is responding with packet loss
else
   if [ "$delay" -lt 100 ] ; then
      echo $1 is responding normally
   else
      echo $1 is responding slow
   fi
fi
}

pingtest router
pingtest pc1
pingtest pc2
pingtest pc3
pingtest pc4
pingtest pc5
pingtest server
```

By simply moving the script commands into the "pingtest" subroutine, the maintenance of the script is greatly simplified. The subroutine itself is defined in the beginning with its name followed by parenthesis (pingtest()). The subroutine commands are contained between the curly braces ({…}). The script lines for the subroutine are simply moved in between the subroutine curly braces, and the specific $router variable is replaced with the more generic $1 variable. When the subroutine is invoked by name "pingtest", the following word on the line is passed to the subroutine in variable $1. Thus, the first ping test (pingtest router)

executes the subroutine lines using "router" in $1. The second ping test (pingtest pc1) reuses the same code, this time using "pc1" in $1.

The above script will produce this quick summary of network status:

```
router is responding normally
pc1 is not responding at all
pc2 is responding normally
pc3 is responding normally
pc4 is responding normally
pc5 is responding normally
server is responding normally
```

The list of *pingtest* commands at the end of the script is now much easier to maintain. More tests can be added simply, without needing to understand the inner workings of the subroutine. This maintainability is key to making such scripts truly reusable. This script could be quickly copied into a new environment and edited with the machine names for that environment to make it functional there.

7. Enhancing the Tool

One of the benefits of using subroutines and maintainable code is that it makes the script easy to modify. This means it can be reused easily in new environments. It also means that it can be quickly enhanced to incorporate new ideas. Any capability added to the subroutine will be consistently applied to all checks in the script.

For example, it would be easy to add logging to the pingtest subroutine with an additional line following each echo line. The `*date*` command will attach a time-stamp to each record, and the ">>" appending redirector will send the output to the end of the "report.log" file. A logging line can be inserted after each echo that prints to the screen:

```
echo $1 is not responding at all
echo `date` $1 is not responding at all >>report.log
```

It is also possible to set up the script to automatically run at regular intervals and page someone if anything is detected. The *crontab* command can be used to set up the script to run at every ten minutes (0, 10, 20, 30, 40, and 50 minutes after every hour, every day, every month, any weekday). It will invoke the default text editor to allow you to edit the *crontab* file.

- `crontab -e`

```
0,10,20,30,40,50 * * * * /usr/local/bin/report
```

Instead of displaying the error to the screen, it could notify an on-call pager. For this function, the echo commands would be redirected to the e-mail program

which sends a message to a network-enabled pager. (Of course, if the network is completely down, the message won't get through!)

```
echo router is not responding at all | mail –s'Report' pager@192.168.1.254
```

It also is possible to reduce the screen clutter and use color-codes to represent the status of each device. The *tput* command will look up the specific escape codes required to display colored text for your particular type of terminal. These can then be stored in variables for easy use. (The TERM variable must be correctly set to identify the terminal type. Try typing "TERM=linux" at the command prompt if the colors do not appear correctly.)

```
red=`tput bold;tput setaf 1`     # bright red text
yellow=`tput bold;tput setaf 3`  # bright yellow text
green=`tput setaf 2`             # dim green text
blue=`tput bold;tput setaf 4`    # bright blue text
purple=`tput bold;tput setaf 5`  # bright magenta text
cyan=`tput bold;tput setaf 6`    # bright cyan text
normal=`tput sgr0`               # normal text
```

Then the *echo* commands could be modified to display color codes instead of longer messages. Red could be used for devices that are not responding, yellow for devices that are losing packets, blue for devices that are slow, and green for devices that are OK. The "-n" option on the *echo* command will leave the cursor at its position on the current line so that subsequent text can be echoed onto the same line later. (This needs to specifically include a trailing space within the single-quotes so there will be spaces between the words.) This allows the script to output a single color-coded word for each device as shown here:

```
echo -n $red$1$normal' '          # not responding at all
```

The output of the report would then be greatly simplified down to a single color-coded line. In our example, the names of all the machines will show up as green for good, except for pc1, which shows up as red because it is not responding:

```
➢ report
router PC1 pc2 pc3 pc4 pc5 server
```

One enhancement could be to create html output to display the status screen on a webpage instead of a terminal console. All that would be required would be to put html tags in the *echo* commands, and redirect the results to a webpage file. It would also be helpful to redirect the results to a temporary file until it is complete and then move the completed temporary file to the webpage file quickly. In the example below, the $$ variable is used to append a unique process-id number to the temporary filename so different users running the script at the same time will use different temporary files without interfering with each other.

```
echo '<html><head></head><body>' >> /tmp/report.$$          # start
…
echo '<p style="color: red">pc1</p>' >> /tmp/report.$$      # errors
```

```
…
echo '</body></html>' >> /tmp/report.$$                    # end
mv –f /tmp/report.$$ /website/report.html                 # file
```

Another enhancement could be to use the *arp* command to verify the hardware address of the machine being pinged. A misconfigured PC could be using the same IP as the router. This would cause false responses to be received from that IP address even when the router is down. It would be a simple matter to use the *arp* command to verify that the correct hardware is answering the pingtest. Once this is added to the subroutine, it applies to all machines checked with the pingtest. All that would be required is a second parameter to the pingtest subroutine that specified what hardware address is expected:

```
pingtest router 12:34:56:78:9a:bc
```

The *arp* check would then be added to the pingtest subroutine. It could use the purple color to indicate that the wrong hardware is answering. The first line below performs the *arp* command, grabs the last line, and cuts out the address. The *if-then* statement tests the $arp variable to see if it looks like a proper hardware address with colons in the expected places. If it doesn't it is set to the expected result so that no error is reported. This is because the *arp* cannot get the hardware address in all cases, such as through a router or firewall. The script must determine the difference between getting the wrong address and not getting an address at all. This keeps it generically reusable in all cases. If it does get a hardware address and it doesn't match the expected value, then it would display a purple color for this machine on the screen:

```
arp=`arp $1|tail -1|cut -c34-50`
if [ "${arp:2:1}${arp:5:1}${arp:8:1}${arp:11:1}${arp:14:1}" != ":::::" ] ; then
   arp="$2"
fi

if [ "$arp" != "$2" ] ; then
   echo -n $purple$1$normal' '   # responding with wrong hardware
   echo `date` $1 is responding with wrong hardware >>report.log
elif [ "$loss" = "100%" ] ; then
# etc. continuing with the rest of the pingtest subroutine…
```

8. Applying the Tool to Security Requirements

So far, the *ping* command has been used for most of the examples. This is because it is such a simple command, is available on all platforms, and provides a variety of information that needs to be captured and interpreted. However, any command under Linux can be automated in a similar manner. This section of the paper demonstrates how to apply the generic script to a variety of security situations.

To automate these commands, consider an even more generic subroutine called "check". This subroutine merely compares the actual results of a command with

the desired results, and outputs the name of the check in either green for good or red for bad. This simple subroutine would look like this:

```
check() {
   if [ "$2" != "" –a “$2” $3 ] ; then
      echo -n $green$1$normal' ' # expected result
   else
      echo -n $red$1$normal' '   # unexpected results
      echo `date` $1 was not $3 >>report.log
   fi
}
```

This subroutine would be invoked with a simple one-line call. The first parameter would be the name of the check, which would be the colored word that appears on the screen. The second parameter would be the command to execute contained in between reverse quotes (`…`). This actually executes the command in between the reverse quotes and passes the result to the subroutine. The third parameter is the expected result including the *test* command operator, which would be “=” for equals, “!=” for not equals, “-lt” for less than, and “-gt” for greater than. More information about the *test* command can be found in the manual page invoked with the command *“man test”.*

The generic check subroutine can automate various Linux commands to monitor specific security requirements, using the same techniques already discussed. Any command can be automated into the script and its output interpreted. The *head* command can be used to extract a specified number of lines from the beginning of the output. The *tail* command can be used to extract a specified number of lines from the end. The *grep* (get regular expression and print) command can be used to search for keywords and extract those lines. The *cut* command can be used to cut out specific fields out of the lines extracted. The *wc* (word count) command can be used to count lines to see if the desired number of lines was found. Even more complicated editing and manipulation of output could be performed with the *awk* command.

Server configuration change control can be checked with various Linux commands:

```
echo Server configuration:
check hostname `hostname -s` '= server'
check domain `hostname -d` '= domain.com'
check ipaddress `hostname -i|cut -d" " -f1` '= 192.168.1.254'
check gateway `netstat -nr|grep ^0.0.0.0|cut -c17-27` '= 192.168.1.1'
echo
```

Performance can be checked with *vmstat* and *df* commands:

```
echo Server performance:
check user-cpu `vmstat 1 2|tail -1|cut -c69-71` '-lt 50'
check system-cpu `vmstat 1 2|tail -1|cut -c73-75` '-lt 50'
check idle-cpu `vmstat 1 2|tail -1|cut -c77-79` '-gt 50'
check diskspace `df | head -2|tail -1|cut -c52-54` '-lt 90'
echo
```

Availability of server services can be checked with the *netstat* command:

```
echo Availability of Services:
check afp-over-tcp `netstat -ltu|grep afpovertcp|wc -l` '-gt 0'
check internet-printer `netstat -ltu|grep ipp|wc -l` '-gt 0'
check ssh `netstat -ltu|grep ssh|wc -l` '-gt 0'
check syslog `netstat -ltu|grep syslog|wc -l` '-gt 0'
echo
```

Availability of other network nodes can be checked with the pingtest subroutine as already indicated:

```
echo Availability of Machines:
pingtest router 12:34:56:78:9a:bc
pingtest pc1 12:34:56:78:9a:bd
pingtest pc2 12:34:56:78:9a:be
pingtest pc3 12:34:56:78:9a:bf
pingtest pc4 12:34:56:78:9a:c1
pingtest pc5 12:34:56:78:9a:c2
pingtest server 12:34:56:78:9a:c3
echo
```

File integrity can be checked with the *md5sum* command to detect unauthorized changes. The MD5 message-digest algorithm is defined by RFC1321 (http://www.faqs.org/rfcs/rfc1321.html) as an algorithm that "takes as input a message of arbitrary length and produces as output a 128-bit 'fingerprint' or 'message digest' of the input. The RFC further states that "it is conjectured that it is computationally infeasible to produce two messages having the same message digest." This makes it excellent so summarize a file or command output into a hex string that can be verified to remain unchanged:

```
echo Integrity of Files:
check hostsfile `md5sum /etc/hosts|grep d673d19596a31509157b5c89c3ca11ec |wc -l` '= 1'
check passwd `md5sum /etc/passwd|grep ccad8e19f9c3ba3e56876723c92f314c |wc -l` '= 1'
check inetd.conf `md5sum /etc/inetd.conf|grep 398b450e5322855cc503f9ebc5a0444a |wc -l` '= 1'
echo
```

Unauthorized web changes or defacement can be detected using the *lynx* command to dump out a text-only copy of a webpage and then using md5sum. The Lynx Users Guide v.2.8.3 (http://lynx.isc.org/release/lynx2-8-3/lynx_help/Lynx_users_guide.html) describes lynx as "a fully-featured World Wide Web (WWW) client" that can dump "files on remote systems running *http, gopher, ftp, wais, nntp, finger,* or *cso/ph/qi* servers, and services accessible via logins to telnet, tn3270 or rlogin accounts." Note that in the example, the secondary output for error messages is merged in with the standard output using the "2>&1" redirector. This forces error messages to go into the regular output stream instead of directly to the screen. It wouldn't hurt to do this on all check subroutine calls if it is not clear when it is needed. (Note that the longer lines wrap around onto the next line on this page, but they should be typed into a single line in the script.)

```
echo Integrity of Website:
check www/index.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 0d60f6cd602d1e7eb31c4e57df8a6bac'
```

```
      check www/home.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= b705be7b7ca33e0a169a2fe0f5cc3a08'
      check www/header.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 4de3b88cb46e24193cca011c152ace79'
      check www/footer.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 105a15e76cb75ea74a3e50745aaf20d0'
      check www/menu.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 945d1b1ab846a1f5cbd3a6064be2b413'
      check www/search.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 88253e672ec15b86e2278a9f879d9562'
      check www/page1.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 8969a68b5ea7521831c1bf76e2e58c84'
      check www/page2.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= dd9a044aa4762cdab4b4ebe8508693d9'
      check www/page3.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 92372dc46db0fec87fbf8c779ffd792d'
      echo
```

Incoming port scans or other attempts can be monitored through a web-enabled device, such as a Linksys router using the *lynx* command to access the data via the web interface. A port scan is described by searchSecurity (http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci214054,00.html) as "a series of messages sent by someone attempting to break into a computer to learn which computer network services, each associated with a 'well-known' port number, the computer provides." The example will detect such attempts in the web-enabled log of a Linksys router:

```
echo Incoming attempts:
lynx -auth user:password -dump http://192.168.1.1/inLogTable.htm > /tmp/linksys.log 2>&1
check telnet `grep \ 23$ /tmp/linksys.log|wc -l` '= 0'
check ftp `grep \ 21$ /tmp/linksys.log|wc -l` '= 0'
check ssh `grep \ 22$ /tmp/linksys.log|wc -l` '= 0'
check smtp `grep \ 25$ /tmp/linksys.log|wc -l` '= 0'
check dns `grep \ 53$ /tmp/linksys.log|wc -l` '= 0'
check tftp `grep \ 69$ /tmp/linksys.log|wc -l` '= 0'
check finger `grep \ 79$ /tmp/linksys.log|wc -l` '= 0'
check http `grep \ 80$ /tmp/linksys.log|wc -l` '= 0'
check pop2 `grep \ 109$ /tmp/linksys.log|wc -l` '= 0'
check pop3 `grep \ 110$ /tmp/linksys.log|wc -l` '= 0'
check rpc `grep \ 111$ /tmp/linksys.log|wc -l` '= 0'
check nntp `grep \ 119$ /tmp/linksys.log|wc -l` '= 0'
check ntp `grep \ 123$ /tmp/linksys.log|wc -l` '= 0'
check imap `grep \ 143$ /tmp/linksys.log|wc -l` '= 0'
check netbios-name `grep \ 137$ /tmp/linksys.log|wc -l` '= 0'
check netbios-datagram `grep \ 138$ /tmp/linksys.log|wc -l` '= 0'
check netbios-session `grep \ 139$ /tmp/linksys.log|wc -l` '= 0'
check snmp `grep \ 161$ /tmp/linksys.log|wc -l` '= 0'
check snmptrap `grep \ 162$ /tmp/linksys.log|wc -l` '= 0'
check bgp `grep \ 179$ /tmp/linksys.log|wc -l` '= 0'
check ldap `grep \ 389$ /tmp/linksys.log|wc -l` '= 0'
check ssl `grep \ 443$ /tmp/linksys.log|wc -l` '= 0'
check rexec `grep \ 512$ /tmp/linksys.log|wc -l` '= 0'
check rlogin `grep \ 513$ /tmp/linksys.log|wc -l` '= 0'
check rshell `grep \ 514$ /tmp/linksys.log|wc -l` '= 0'
check lpd `grep \ 515$ /tmp/linksys.log|wc -l` '= 0'
check kazaa `grep \ 1214$ /tmp/linksys.log|wc -l` '= 0'
check ms-sql `grep \ 1433$ /tmp/linksys.log|wc -l` '= 0'
check nfs `grep \ 2049$ /tmp/linksys.log|wc -l` '= 0'
check lockd `grep \ 4045$ /tmp/linksys.log|wc -l` '= 0'
check x-windows `grep \ 6000$ /tmp/linksys.log|wc -l` '= 0'
check gnutella `grep \ 6349$ /tmp/linksys.log|wc -l` '= 0'
check proxy `grep \ 8080$ /tmp/linksys.log|wc -l` '= 0'
echo
```

Trojan worm attacks and common backdoor scans can be detected using the same technique described above. Robert Vamosi, in his article "Will 2002 be the year of the Trojan Horse?" (http://hwreviews.netscape.com/techtrends/0-6014-8-8724341-1.html) calls 2001 "The Year of the Worm" and predicts 2002 to be "The Year of the Trojan Horse," and he predicts it will get worse. The example detects incoming worm attacks or attempts by hackers to reach backdoor ports used by Trojan horses:

```
echo Incoming Trojans and Worms:
check Happy99 `grep \ 119$ /tmp/linksys.log|wc -l` '= 0'
check NetTaxi `grep \ 142$ /tmp/linksys.log|wc -l` '= 0'
check Incognito `grep \ 420$ /tmp/linksys.log|wc -l` '= 0'
check 666 `grep \ 666$ /tmp/linksys.log|wc -l` '= 0'
check Millenium-Worm `grep \ 1338$ /tmp/linksys.log|wc -l` '= 0'
check SocketsdeTroie `grep \ 5001$ /tmp/linksys.log|wc -l` '= 0'
check NetBus-Worm `grep \ 6666$ /tmp/linksys.log|wc -l` '= 0'
check SubZero `grep \ 15382$ /tmp/linksys.log|wc -l` '= 0'
check SubSeven `grep \ 16959$ /tmp/linksys.log|wc -l` '= 0'
check Millenium `grep \ 20000$ /tmp/linksys.log|wc -l` '= 0'
check NetTrojan `grep \ 29104$ /tmp/linksys.log|wc -l` '= 0'
check Back-Oriface `grep \ 31337$ /tmp/linksys.log|wc -l` '= 0'
check NetSpy `grep \ 31339$ /tmp/linksys.log|wc -l` '= 0'
echo
```

SNMP devices can be accessed with commands such as *snmpwalk*. The IETF (Internet Engineering Task Force) has a working group for configuration management with SNMP (simple network management protocol) called snmpconf (http://www.ietf.org/html.charters/snmpconf-charter.html) which "will create a Best Current Practices" document which outlines the most effective methods for using the SNMP Framework to accomplish configuration management."

```
echo Router interface1 snmp:
check index `snmpwalk router public interfaces.ifTable.ifEntry.ifIndex.1
|cut -d" " -f3` '= 1'
check descr `snmpwalk router public interfaces.ifTable.ifEntry.ifDescr.1
|cut -d" " -f3` '= K32_MAC'
check type `snmpwalk router public interfaces.ifTable.ifEntry.ifType.1
|cut -d" " -f3` '= ethernetCsmacd(6)'
check mtu `snmpwalk router public interfaces.ifTable.ifEntry.ifMtu.1|cut
-d" " -f3` '= 1500'
check gauge32 `snmpwalk router public interfaces.ifTable.ifEntry.ifSpeed.1 |cut -d" "
-f4` '= 10000000'
check physaddress `snmpwalk router public interfaces.ifTable.ifEntry.ifPhysAddress.1
|cut -d" " -f3` '= 12:34:56:78:9a:bc'
check adminstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifAdminStatus.1
|cut -d" " -f3` '= up(1)'
check operstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifOperStatus.1
|cut -d" " -f3` '= up(1)'
echo

echo Router interface2 snmp:
check index `snmpwalk router public interfaces.ifTable.ifEntry.ifIndex.2
|cut -d" " -f3` '= 2'
check descr `snmpwalk router public interfaces.ifTable.ifEntry.ifDescr.2
|cut -d" " -f3` '= NE2000'
check type `snmpwalk router public interfaces.ifTable.ifEntry.ifType.2
|cut -d" " -f3` '= ethernetCsmacd(6)'
check mtu `snmpwalk router public interfaces.ifTable.ifEntry.ifMtu.2|cut
-d" " -f3` '= 1492'
check gauge32 `snmpwalk router public interfaces.ifTable.ifEntry.ifSpeed.2 |cut -d" "
-f4` '= 10000000'
```

```
        check physaddress `snmpwalk router public interfaces.ifTable.ifEntry.ifPhysAddress.2
|cut -d" " -f3` '= 12:34:56:78:9a:bb'
        check adminstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifAdminStatus.2
|cut -d" " -f3` '= up(1)'
        check operstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifOperStatus.2
|cut -d" " -f3` '= up(1)'
        echo
```

Unauthorized servers on the network can be detected with the *nmap* command or other port scanners. The *nmap* home page (http://www.insecure.org/nmap/index.html) describes *nmap* as "an open source utility for network exploration or security auditing" which can detect hosts, services, OS version, firewall filters, and "dozens of other characteristics."

```
        echo Portscan:
        check pc1:ftp `nmap -p21 pc1|grep open|wc -l` '= 0'
        check pc2:ftp `nmap -p21 pc2|grep open|wc -l` '= 0'
        check pc3:ftp `nmap -p21 pc3|grep open|wc -l` '= 0'
        check pc4:ftp `nmap -p21 pc4|grep open|wc -l` '= 0'
        check pc5:ftp `nmap -p21 pc5|grep open|wc -l` '= 0'
        check pc1:ssh `nmap -p22 pc1|grep open|wc -l` '= 0'
        check pc2:ssh `nmap -p22 pc2|grep open|wc -l` '= 0'
        check pc3:ssh `nmap -p22 pc3|grep open|wc -l` '= 0'
        check pc4:ssh `nmap -p22 pc4|grep open|wc -l` '= 0'
        check pc5:ssh `nmap -p22 pc5|grep open|wc -l` '= 0'
        check pc1:telnet `nmap -p23 pc1|grep open|wc -l` '= 0'
        check pc2:telnet `nmap -p23 pc2|grep open|wc -l` '= 0'
        check pc3:telnet `nmap -p23 pc3|grep open|wc -l` '= 0'
        check pc4:telnet `nmap -p23 pc4|grep open|wc -l` '= 0'
        check pc5:telnet `nmap -p23 pc5|grep open|wc -l` '= 0'
        check pc1:smtp `nmap -p25 pc1|grep open|wc -l` '= 0'
        check pc2:smtp `nmap -p25 pc2|grep open|wc -l` '= 0'
        check pc3:smtp `nmap -p25 pc3|grep open|wc -l` '= 0'
        check pc4:smtp `nmap -p25 pc4|grep open|wc -l` '= 0'
        check pc5:smtp `nmap -p25 pc5|grep open|wc -l` '= 0'
        echo
```

Specific version levels of software products can be monitored using the *netcat* command to query specific ports on remote machines and interpreting the results. The Debian package for *netcat* (http://packages.debian.org/testing/net/netcat.html) describes netcat as "a simple Unix utility which reads and writes data across network connections using TCP or UDP protocol." The package also notes that "it is designed to be a reliable 'back-end' tool that can be used directly or easily driven by other programs and scripts." It is a "feature-rich network debugging and exploration tool, since it can create almost any kind of connection…." Note that in the example, the commands inside the reverse-quotes (`…`) might return a null value, so the entire thing is enclosed in double-quotes ("…") so that there isn't a missing parameter for the subroutine call. It wouldn't hurt to do this on all check subroutine calls if it is not clear when it is needed.

```
        echo SSH-versions:
        check pc1-ssh-version "`echo QUIT|netcat -w1 pc1 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc2-ssh-version "`echo QUIT|netcat -w1 pc2 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc3-ssh-version "`echo QUIT|netcat -w1 pc3 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc4-ssh-version "`echo QUIT|netcat -w1 pc4 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
```

```
      check pc5-ssh-version "`echo QUIT|netcat -w1 pc5 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
      echo
```

Log analysis can be performed by using simple Linux commands to parse logs for items of interest. The /var/log directory contains the primary "messages" file. The cups subdirectory contains printer logs if the server is using the Common Linux Printing System. The httpd subdirectory contains various web logs if the server is running a website. The ippl subdirectory contains IP protocol logs if the server is running it. The ircd subdirectory contains irc logs if the server is running an irc chat room. The news subdirectory contains news logs if the server is relaying usenet news. There are many combinations of events that can be extracted from these various log files, and they will be different for every site. Here are some various examples of log monitoring:

```
echo Log Monitoring:
check keyboarderrors `grep keyboard /var/log/messages|wc -l` '= 0'
check ssherrors `grep sshd /var/log/messages|wc -l` '= 0'
check newprinters `grep New\ printer /var/log/cups/error_log|wc -l` '= 0'
echo
```

Logging can also be achieved by simply recording results with timestamps into a file. This has already been demonstrated in the report script. The *grep* command can extract specific types of log entries to produce various specific logs. The following commands show what can be extracted from the report.log file that is created by the report script:

```
grep router report.log          # log of router status records
grep pc1 report.log             # log of pc1 status records
grep user-cpu report.log        # log of user cpu usage
grep diskspace report.log       # log of disk space usage
grep www/index.html report.log  # log of website changes
grep finger report.log          # log of finger attempts
grep Millenium-worm report.log  # log of Millenium worm activity
grep :ftp report.log            # log of unauthorized ftp servers
grep ssh-version report.log     # log of ssh version levels
```

9. Conclusions

This should be enough information for security professionals to start creating their own generic reusable Linux scripts. Basic Linux scripting can be used to create tools quickly and easily with little cost. It is important to remember to always make scripts reusable and generic. Otherwise, they will have to be rebuilt from scratch every time. The more maintainable a script is, the more likely it is to be adapted to changing requirements and ported to new environments. More advanced scripting languages are available than the bash shell script used in this paper, but it is important to consider how widely available they may be in future environments.

It is suggested that all security professionals maintain their own reusable library of Linux scripts. Not only will these speed up future projects, but they will also

provide a consistency and quality across all projects. Any good ideas that are implemented into the script can be reused on later projects. Minor details or enhancements will not be forgotten. Having one's own resources can distinguish a professional from other practitioners who all have the same toolsets. These resources can also enable allow the professional to remain fully functional in an environment that is lacking in tools. This allows the security professional to operate independently and effectively without having to rely on the provisions of others.

Appendix A: Complete Example Script

```
#!/bin/bash
# "report" script to report network status

red=`tput setaf 1`  # bright red text
yellow=`tput setaf 3`      # bright yellow text
green=`tput setaf 2`       # dim green text
blue=`tput setaf 4` # bright blue text
purple=`tput setaf 5`      # bright magenta text
cyan=`tput setaf 6` # bright cyan text
normal=`tput sgr0`  # normal text

pingtest() {
   ping=`ping -c3 $1 | tail -2`
   loss=`echo $ping | cut -d"," -f3 | cut -d" " -f2`
   delay=`echo $ping | cut -d"=" -f2 | cut -d"." -f1`
   arp=`arp $1|tail -1|cut -c34-50`
   if [ "${arp:2:1}${arp:5:1}${arp:8:1}${arp:11:1}${arp:14:1}" != ":::::" ] ; then
      arp="$2"
   fi

   if [ "$arp" != "$2" ] ; then
      echo -n $purple$1$normal' '# responding with wrong hardware
      echo `date` $1 is responding with wrong hardware >>report.log
   elif [ "$loss" = "100%" ] ; then
      echo -n $red$1$normal' '   # not responding at all
      echo `date` $1 is not responding at all >>report.log
   elif [ "$loss" != "0%" ] ; then
      echo -n $yellow$1$normal' '# responding with packet loss
      echo `date` $1 is responding with packet loss >>report.log
   else
      if [ "$delay" -lt 100 ] ; then
         echo -n $green$1$normal' '    # not responding slow
         echo `date` $1 is not slow >>report.log
      else
         echo -n $blue$1$normal' '     # responding slow
         echo `date` $1 is slow >>report.log
      fi
   fi

}

check() {
   if [ "$2" != "" -a "$2" $3 ] ; then
      echo -n $green$1$normal' '       # expected result
   else
      echo -n $red$1$normal' '         # unexpected results
      echo `date` $1 was not $3 >>report.log
   fi
}

echo Server configuration:
check hostname `hostname -s` '= server'
check domain `hostname -d` '= domain.com'
check ipaddress `hostname -i|cut -d" " -f1` '= 192.168.1.254'
check gateway `netstat -nr|grep ^0.0.0.0|cut -c17-27` '= 192.168.1.1'
echo

echo Server performance:
check user-cpu `vmstat 1 2|tail -1|cut -c69-71` '-lt 50'
check system-cpu `vmstat 1 2|tail -1|cut -c73-75` '-lt 50'
check idle-cpu `vmstat 1 2|tail -1|cut -c77-79` '-gt 50'
check diskspace `df | head -2|tail -1|cut -c52-54` '-lt 90'
echo

echo Availability of Services:
check afp-over-tcp `netstat -ltu|grep afpovertcp|wc -l` '-gt 0'
check internet-printer `netstat -ltu|grep ipp|wc -l` '-gt 0'
```

```
        check ssh `netstat -ltu|grep ssh|wc -l` '-gt 0'
        check syslog `netstat -ltu|grep syslog|wc -l` '-gt 0'
        echo

        echo Availability of Machines:
        pingtest router 12:34:56:78:9a:bc
        pingtest pc1 12:34:56:78:9a:bd
        pingtest pc2 12:34:56:78:9a:be
        pingtest pc3 12:34:56:78:9a:bf
        pingtest pc4 12:34:56:78:9a:c1
        pingtest pc5 12:34:56:78:9a:c2
        pingtest server 12:34:56:78:9a:c3
        echo

        echo Integrity of Files:
        check hostsfile `md5sum /etc/hosts|grep d673d19596a31509157b5c89c3ca11ec |wc -l` '= 1'
        check passwd `md5sum /etc/passwd|grep ccad8e19f9c3ba3e56876723c92f314c |wc -l` '= 1'
        check inetd.conf `md5sum /etc/inetd.conf|grep 398b450e5322855cc503f9ebc5a0444a |wc -l` '= 1'
        echo

        echo Integrity of Website:
        check www/index.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 0d60f6cd602d1e7eb31c4e57df8a6bac'
        check www/home.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= b705be7b7ca33e0a169a2fe0f5cc3a08'
        check www/header.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 4de3b88cb46e24193cca011c152ace79'
        check www/footer.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 105a15e76cb75ea74a3e50745aaf20d0'
        check www/menu.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 945d1b1ab846a1f5cbd3a6064be2b413'
        check www/search.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 88253e672ec15b86e2278a9f879d9562'
        check www/page1.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 8969a68b5ea7521831c1bf76e2e58c84'
        check www/page2.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= dd9a044aa4762cdab4b4ebe8508693d9'
        check www/page3.html `lynx -reload -dump http://192.168.1.253 2>&1 |md5sum
|cut -d" " -f1` '= 92372dc46db0fec87fbf8c779ffd792d'
        echo

        echo Incoming attempts:
        lynx -auth user:password -dump http://192.168.1.1/inLogTable.htm > /tmp/linksys.log 2>&1
        check telnet `grep \ 23$ /tmp/linksys.log|wc -l` '= 0'
        check ftp `grep \ 21$ /tmp/linksys.log|wc -l` '= 0'
        check ssh `grep \ 22$ /tmp/linksys.log|wc -l` '= 0'
        check smtp `grep \ 25$ /tmp/linksys.log|wc -l` '= 0'
        check dns `grep \ 53$ /tmp/linksys.log|wc -l` '= 0'
        check tftp `grep \ 69$ /tmp/linksys.log|wc -l` '= 0'
        check finger `grep \ 79$ /tmp/linksys.log|wc -l` '= 0'
        check http `grep \ 80$ /tmp/linksys.log|wc -l` '= 0'
        check pop2 `grep \ 109$ /tmp/linksys.log|wc -l` '= 0'
        check pop3 `grep \ 110$ /tmp/linksys.log|wc -l` '= 0'
        check rpc `grep \ 111$ /tmp/linksys.log|wc -l` '= 0'
        check nntp `grep \ 119$ /tmp/linksys.log|wc -l` '= 0'
        check ntp `grep \ 123$ /tmp/linksys.log|wc -l` '= 0'
        check imap `grep \ 143$ /tmp/linksys.log|wc -l` '= 0'
        check netbios-name `grep \ 137$ /tmp/linksys.log|wc -l` '= 0'
        check netbios-datagram `grep \ 138$ /tmp/linksys.log|wc -l` '= 0'
        check netbios-session `grep \ 139$ /tmp/linksys.log|wc -l` '= 0'
        check snmp `grep \ 161$ /tmp/linksys.log|wc -l` '= 0'
        check snmptrap `grep \ 162$ /tmp/linksys.log|wc -l` '= 0'
        check bgp `grep \ 179$ /tmp/linksys.log|wc -l` '= 0'
        check ldap `grep \ 389$ /tmp/linksys.log|wc -l` '= 0'
        check ssl `grep \ 443$ /tmp/linksys.log|wc -l` '= 0'
        check rexec `grep \ 512$ /tmp/linksys.log|wc -l` '= 0'
        check rlogin `grep \ 513$ /tmp/linksys.log|wc -l` '= 0'
        check rshell `grep \ 514$ /tmp/linksys.log|wc -l` '= 0'
        check lpd `grep \ 515$ /tmp/linksys.log|wc -l` '= 0'
        check kazaa `grep \ 1214$ /tmp/linksys.log|wc -l` '= 0'
        check ms-sql `grep \ 1433$ /tmp/linksys.log|wc -l` '= 0'
        check nfs `grep \ 2049$ /tmp/linksys.log|wc -l` '= 0'
```

```
        check lockd `grep \ 4045$ /tmp/linksys.log|wc -l` '= 0'
        check x-windows `grep \ 6000$ /tmp/linksys.log|wc -l` '= 0'
        check gnutella `grep \ 6349$ /tmp/linksys.log|wc -l` '= 0'
        check proxy `grep \ 8080$ /tmp/linksys.log|wc -l` '= 0'

        echo

        echo Incoming Trojans and Worms:
        check Happy99 `grep \ 119$ /tmp/linksys.log|wc -l` '= 0'
        check NetTaxi `grep \ 142$ /tmp/linksys.log|wc -l` '= 0'
        check Incognito `grep \ 420$ /tmp/linksys.log|wc -l` '= 0'
        check 666 `grep \ 666$ /tmp/linksys.log|wc -l` '= 0'
        check Millenium-Worm `grep \ 1338$ /tmp/linksys.log|wc -l` '= 0'
        check SocketsdeTroie `grep \ 5001$ /tmp/linksys.log|wc -l` '= 0'
        check NetBus-Worm `grep \ 6666$ /tmp/linksys.log|wc -l` '= 0'
        check SubZero `grep \ 15382$ /tmp/linksys.log|wc -l` '= 0'
        check SubSeven `grep \ 16959$ /tmp/linksys.log|wc -l` '= 0'
        check Millenium `grep \ 20000$ /tmp/linksys.log|wc -l` '= 0'
        check NetTrojan `grep \ 29104$ /tmp/linksys.log|wc -l` '= 0'
        check Back-Oriface `grep \ 31337$ /tmp/linksys.log|wc -l` '= 0'
        check NetSpy `grep \ 31339$ /tmp/linksys.log|wc -l` '= 0'
        echo

        echo Router interface1 snmp:
        check index `snmpwalk router public interfaces.ifTable.ifEntry.ifIndex.1
|cut -d" " -f3` '= 1'
        check descr `snmpwalk router public interfaces.ifTable.ifEntry.ifDescr.1
|cut -d" " -f3` '= K32_MAC'
        check type `snmpwalk router public interfaces.ifTable.ifEntry.ifType.1
|cut -d" " -f3` '= ethernetCsmacd(6)'
        check mtu `snmpwalk router public interfaces.ifTable.ifEntry.ifMtu.1|cut
-d" " -f3` '= 1500'
        check gauge32 `snmpwalk router public interfaces.ifTable.ifEntry.ifSpeed.1 |cut -d" "
-f4` '= 10000000'
        check physaddress `snmpwalk router public interfaces.ifTable.ifEntry.ifPhysAddress.1
|cut -d" " -f3` '= 12:34:56:78:9a:bc'
        check adminstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifAdminStatus.1
|cut -d" " -f3` '= up(1)'
        check operstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifOperStatus.1
|cut -d" " -f3` '= up(1)'
        echo

        echo Router interface2 snmp:
        check index `snmpwalk router public interfaces.ifTable.ifEntry.ifIndex.2
|cut -d" " -f3` '= 2'
        check descr `snmpwalk router public interfaces.ifTable.ifEntry.ifDescr.2
|cut -d" " -f3` '= NE2000'
        check type `snmpwalk router public interfaces.ifTable.ifEntry.ifType.2
|cut -d" " -f3` '= ethernetCsmacd(6)'
        check mtu `snmpwalk router public interfaces.ifTable.ifEntry.ifMtu.2|cut
-d" " -f3` '= 1492'
        check gauge32 `snmpwalk router public interfaces.ifTable.ifEntry.ifSpeed.2 |cut -d" "
-f4` '= 10000000'
        check physaddress `snmpwalk router public interfaces.ifTable.ifEntry.ifPhysAddress.2
|cut -d" " -f3` '= 12:34:56:78:9a:bb'
        check adminstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifAdminStatus.2
|cut -d" " -f3` '= up(1)'
        check operstatus `snmpwalk router public interfaces.ifTable.ifEntry.ifOperStatus.2
|cut -d" " -f3` '= up(1)'
        echo

        echo Portscan:
        check pc1:ftp `nmap -p21 pc1|grep open|wc -l` '= 0'
        check pc2:ftp `nmap -p21 pc2|grep open|wc -l` '= 0'
        check pc3:ftp `nmap -p21 pc3|grep open|wc -l` '= 0'
        check pc4:ftp `nmap -p21 pc4|grep open|wc -l` '= 0'
        check pc5:ftp `nmap -p21 pc5|grep open|wc -l` '= 0'
        check pc1:ssh `nmap -p22 pc1|grep open|wc -l` '= 0'
        check pc2:ssh `nmap -p22 pc2|grep open|wc -l` '= 0'
        check pc3:ssh `nmap -p22 pc3|grep open|wc -l` '= 0'
        check pc4:ssh `nmap -p22 pc4|grep open|wc -l` '= 0'
        check pc5:ssh `nmap -p22 pc5|grep open|wc -l` '= 0'
```

```
        check pc1:telnet `nmap -p23 pc1|grep open|wc -l` '= 0'
        check pc2:telnet `nmap -p23 pc2|grep open|wc -l` '= 0'
        check pc3:telnet `nmap -p23 pc3|grep open|wc -l` '= 0'
        check pc4:telnet `nmap -p23 pc4|grep open|wc -l` '= 0'
        check pc5:telnet `nmap -p23 pc5|grep open|wc -l` '= 0'
        check pc1:smtp `nmap -p25 pc1|grep open|wc -l` '= 0'
        check pc2:smtp `nmap -p25 pc2|grep open|wc -l` '= 0'
        check pc3:smtp `nmap -p25 pc3|grep open|wc -l` '= 0'
        check pc4:smtp `nmap -p25 pc4|grep open|wc -l` '= 0'
        check pc5:smtp `nmap -p25 pc5|grep open|wc -l` '= 0'
        echo

        echo SSH-versions:
        check pc1-ssh-version "`echo QUIT|netcat -w1 pc1 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc2-ssh-version "`echo QUIT|netcat -w1 pc2 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc3-ssh-version "`echo QUIT|netcat -w1 pc3 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc4-ssh-version "`echo QUIT|netcat -w1 pc4 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        check pc5-ssh-version "`echo QUIT|netcat -w1 pc5 22|head -1`" '= SSH-1.99-
OpenSSH_3.4p1'
        echo

        echo Log Monitoring:
        check keyboarderrors `grep keyboard /var/log/messages|wc -l` '= 0'
        check sshderrors `grep sshd /var/log/messages|wc -l` '= 0'
        check newprinters `grep New\ printer /var/log/cups/error_log|wc -l` '= 0'
        echo
```

Appendix B: Complete Example Output

```
Server configuration:
 hostname domain ipaddress gateway
Server performance:
 user-cpu system-cpu idle-cpu diskspace
Availability of Services:
 afp-over-tcp internet-printer ssh syslog
Availability of Machines:
 router PC1 pc2 pc3 pc4 pc5 server
Integrity of Files:
 hostsfile passwd inetd.conf
Integrity of Website:
 www/index.html www/home.html www/header.html www/footer.html www/menu.html
 www/search.html www/page1.html www/page2.html www/page2.html
Incoming attempts:
 telnet ftp ssh smtp dns tftp finger http pop2 pop3 rpc nntp ntp imap netbios-
 name netbios-datagram netbios-session snmp snmptrap bgp ldap ssl rexec rlogin
 rshell lpd kazaa ms-sql nfs lockd x-windows gnutella proxy
Incoming Trojans and Worms:
 Happy99 NetTaxi Incognito 666 Millenium-Worm SocketsdeTroie NetBus-Worm
 SubZero SubSeven Millenium NetTrojan Back-Oriface NetSpy
Router interface1 snmp:
 index descr type mtu gauge32 physaddress adminstatus operstatus
Router interface2 snmp:
 index descr type mtu gauge32 physaddress adminstatus operstatus
Portscan:
 pc1:ftp pc2:ftp pc3:ftp pc4:ftp pc5:ftp
 pc1:ssh pc2:ssh pc3:ssh pc4:ssh pc5:ssh
 pc1:telnet pc2:telnet pc3:telnet pc4:telnet pc5:telnet
 pc1:smtp pc2:smtp pc3:smtp pc4:smtp pc5:smtp
SSH-versions:
 pc1-ssh-version pc2-ssh-version pc3-ssh-version pc4-ssh-version pc5-ssh-version
Log Monitoring:
 keyboarderrors ssherrors newprinters
```

Appendix C: References

Cooper, Mendel. "Why Shell Programming?" *Advanced Bash-Scripting Guide.* Version 1.5. 13 July 2002. URL: http://mirrors.sunsite.dk/ldp/LDP/abs/html/why-shell.html (23 Aug 2002).

Foster, Decklin. "Package: netcat 1.10-22" *Debian Project.* 18 Aug 2002. URL: http://packages.debian.org/testing/net/netcat.html (23 Aug 2002).

Fyodor. "Introduction" *Nmap Free Security Scanner.* 10 Aug 2002. URL: http://www.insecure.org/nmap/index.html (23 Aug 2002).

Internet Engineering Task Force. "Description of Working Group" *Configuration Management with SNMP (snmpconf) Charter.* 21 May 2002. URL: http://www.ietf.org/html.charters/snmpconf-charter.html (23 Aug 2002).

Internet Software Consortium. "Preface" *Lynx User's Guide.* Version 2.8.3. July 2001. URL: http://lynx.isc.org/release/lynx2-8-3/lynx_help/Lynx_users_guide.html (23 Aug 2002).

Parker, Steve. "Philosophy" *Shell Scripting Tutorial.* 9 April 2002. URL: http://steve-parker.org/sh/philosophy.shtml (23 Aug 2002).

Rivest, R. "Executive Summary" *The MD5 Message-Digest Algorithm.* RFC-1321. April 1992. URL: http://www.rfc-editor.org/rfc/rfc1321.txt (23 Aug 2002).

SearchSecurity. "Port Scan" *Definitions.* 31 July 2001. URL: http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci214054,00.html (23 Aug 2002).

Sheer, Paul. "What are Unix systems for? What can Linux do?" *Linux: Rute User's Tutorial and Exposition.* URL: http://rute.sourceforge.net/node51.html (23 Aug 2002).

Vamosi, Robert. "Will 2002 be the Year of the Trojan Horse?" Security Watch. 6 February 2002. URL: http://hwreviews.netscape.com/techtrends/0-6014-8-8724341-1.html (23 Aug 2002).

Last Updated: February 22nd, 2018

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

CyberThreat Summit 2018	London, GB	Feb 27, 2018 - Feb 28, 2018	Live Event
SANS London March 2018	London, GB	Mar 05, 2018 - Mar 10, 2018	Live Event
SANS Secure Osaka 2018	Osaka, JP	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS San Francisco Spring 2018	San Francisco, CAUS	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS Paris March 2018	Paris, FR	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS Secure Singapore 2018	Singapore, SG	Mar 12, 2018 - Mar 24, 2018	Live Event
SANS Northern VA Spring - Tysons 2018	McLean, VAUS	Mar 17, 2018 - Mar 24, 2018	Live Event
ICS Security Summit & Training 2018	Orlando, FLUS	Mar 18, 2018 - Mar 26, 2018	Live Event
SANS Munich March 2018	Munich, DE	Mar 19, 2018 - Mar 24, 2018	Live Event
SEC487: Open-Source Intel Beta One	McLean, VAUS	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Pen Test Austin 2018	Austin, TXUS	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Secure Canberra 2018	Canberra, AU	Mar 19, 2018 - Mar 24, 2018	Live Event
SANS Boston Spring 2018	Boston, MAUS	Mar 25, 2018 - Mar 30, 2018	Live Event
SANS 2018	Orlando, FLUS	Apr 03, 2018 - Apr 10, 2018	Live Event
SANS Abu Dhabi 2018	Abu Dhabi, AE	Apr 07, 2018 - Apr 12, 2018	Live Event
Pre-RSA® Conference Training	San Francisco, CAUS	Apr 11, 2018 - Apr 16, 2018	Live Event
SANS Zurich 2018	Zurich, CH	Apr 16, 2018 - Apr 21, 2018	Live Event
SANS London April 2018	London, GB	Apr 16, 2018 - Apr 21, 2018	Live Event
SANS Baltimore Spring 2018	Baltimore, MDUS	Apr 21, 2018 - Apr 28, 2018	Live Event
SANS Seattle Spring 2018	Seattle, WAUS	Apr 23, 2018 - Apr 28, 2018	Live Event
Blue Team Summit & Training 2018	Louisville, KYUS	Apr 23, 2018 - Apr 30, 2018	Live Event
SANS Riyadh April 2018	Riyadh, SA	Apr 28, 2018 - May 03, 2018	Live Event
SANS Doha 2018	Doha, QA	Apr 28, 2018 - May 03, 2018	Live Event
SANS SEC460: Enterprise Threat Beta Two	Crystal City, VAUS	Apr 30, 2018 - May 05, 2018	Live Event
Automotive Cybersecurity Summit & Training 2018	Chicago, ILUS	May 01, 2018 - May 08, 2018	Live Event
SANS SEC504 in Thai 2018	Bangkok, TH	May 07, 2018 - May 12, 2018	Live Event
SANS Security West 2018	San Diego, CAUS	May 11, 2018 - May 18, 2018	Live Event
SANS Melbourne 2018	Melbourne, AU	May 14, 2018 - May 26, 2018	Live Event
SANS Northern VA Reston Spring 2018	Reston, VAUS	May 20, 2018 - May 25, 2018	Live Event
SANS New York City Winter 2018	OnlineNYUS	Feb 26, 2018 - Mar 03, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced