Global Information Assurance Certification Paper

IOSMap: TCP and UDP Port Scanning on Cisco IOS Platforms

GCIA Gold Certification

Author Robert VandenBrink
Advisor Rick Wanner

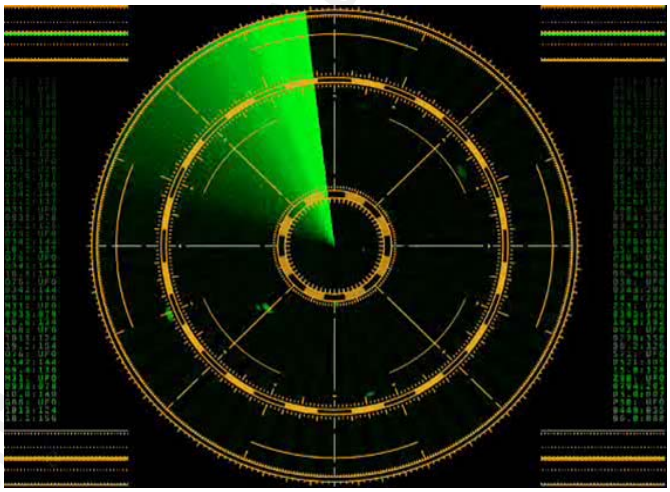


Table of Contents

Introduction

This paper describes IOSmap, a port scanning tool implemented on Cisco IOS using the native TCL (Tool Command Language) scripting language on that platform. The business requirement for this tool, implementation considerations and challenges, and design choices are discussed.

The Business Requirement

Writing a tool like a port scanner to run on Cisco IOS might seem like an unusual approach – some might say it sounds a lot like a solution looking for a problem. However, there are in fact some real-world scenarios where a solution of this type can fill a unique requirement.

One such circumstance is one that is sometimes seen as a security consultant, and involves some specific customer security constraints and requirements. For instance, suppose a company employee is suspected of running a peer-to-peer file distribution application. This violates Corporate Policy, and is illegal in the jurisdiction of the remote location in which the employee is working. The person being investigated is in the IT group, so has access and responsibility for change control on the corporate workstations and servers, and has responsibility for automated tools to report changes on these platforms. Further to this, the security access this employee regularly has is high enough to expose sensitive information, and the fear is that this information is exposed on the peer-to-peer network being used. The final constraint is that Corporate Policy further states that contractor hardware and non-approved software cannot be utilized in engagements of this type, for fear of inadvertent data exposure (via malware), or intentional data exposure.

This scenario in fact did occur, and the solution that was arrived at was to use a non-critical Cisco router (in fact, the router local to the network being investigated) to scan the suspect network for TCP ports commonly used by peer-to-peer file sharing applications. The port scan was scripted using TCL, a scripting language available on most modern routers and higher-end switches. After the initial port scans found suspect ports, the same local router was used to capture actual peer-to-peer traffic to build a body of evidence to take to the Corporate HR Group. After completing this engagement, the primitive port scanner was “cleaned up”, given some help text and transformed into a more general purpose tool (IOSmap) that can be used by others in similar circumstances.

Platform Selection and Caveats

Routing devices are almost always critical components of the infrastructure in any network, large or small. Because of this, it is recommended that wherever possible non-core, passive backup or spare hardware be used when running complex scripts. At all times, the resource constraints of the router should be kept in mind. Routers are typically constrained on memory, but often have CPU cycles to spare. Because of this, a short subroutine was written to estimate the overall impact of the script prior to execution. If the CPU or memory utilization is estimated to be potentially excessive, IOSmap displays an error message, and the user has an opportunity to exit the script immediately. Finally, TCL has no "Ctrl-C" equivalent on IOS, so larger scans that were perhaps started by accident are not easily ended (unless the terminal session is simply exited). For these reasons, IOSmap is most often used for "targeted scans", where a limited number of addresses and/or ports are scanned. Full subnet scans or full range (1-65535) port scans are generally not recommended.

A discussion of operational caveats of tools of this type would not be complete without covering two more points: TCL requires privilege level 15 (full admin) rights to execute a script, and parts of IOSmap will modify the config and/or enable specific debugs. This should be kept in mind when using IOSmap, as change control requests will often be required for each of these 3 actions.

Application Syntax

The input syntax and output format was selected to be as close to standard, familiar tools as possible. To this end, both the inputs and outputs were designed to be similar to the popular NMAP scanning tool. The IOSmap tool is not presented as an NMAP port, it is a simple TCP and UDP port scanner on a constrained platform, so does not have either the capabilities, feature set, flexibility, breadth or speed of the NMAP tool.

All IOSmap parameters are defined at the command line. The help text for IOSmap shows all the scan options available:

```
HOST DISCOVERY:
  -P0  Treat all hosts as online - skip Ping test
  -SL  List hosts and ports to scan
SCAN TYPE:
  -sP  Ping scan only <ICMP ECHO>
  -sT  TCP Connect Scan
  -sU  UDP Scan
  --reason:  display the reason a port state is reported as such
PORT SPECIFICATION:
  -p <port ranges> Specify ports to scan.
     -p22  Scan port 22
     -p22,23,135-139,445  Scan ports 22, 23, 135, 136, 137, 138, 139, 445
TARGET SPECIFICATION:
  CIDR, IP range and single IPs are all a supported - comma delimited
  For example:
    192.168.10.0/24,192.168.17.21-34,192.168.40.1
```

Host Specifcation, Parsing IP's and Ports

Validity Checks

Prior to execution, several checks are made to ensure that inputs are valid. All addresses specified are verified, to ensure that networks are specified with exactly 4 valid octets of 0-255, and that networks specified via CIDR notation are properly specified with bitmasks of 8-30. If the bitmask is less than 8, it is deemed that IOS not a good platform for the scan due to resource utilization, and the scan should be broken up if it was really required. If a bitmask greater than 30 is specified, individual addresses or a short address range is considered a better method to specify the target. Finally, networks or ports in ranges are verified that they are entered low-to-high.

Port Specification, Parsing Ports

Ports are entered as comma separated entities, specified with a "–p" command line argument. Each entity can be a single port, or a group of ports separated by a dash. For instance, the string "-p22-25,135,139" would specify ports 22,23,24,25,135 and 139. A validity check is done before proceeding to ensure that all ports are in the valid range of 1-65535 (scanning for port 0 is not supported).

Scan Types

Scan types can be either TCP, UDP, Ping only or List only.

TCP Connect Port Scanning

Since the TCL implemented on IOS does not permit the formation of raw packets, the only form of TCP scanning that can be realized is a simple TCP connect scan. If no scan type is specified on the IOSmap command line, TCP Scans are the default. TCP scanning can be implicitly specified with a "–sT" command line argument. The table below indicates the port status inferred for each possible return:

Return code to TCP Connect request	**Port Status**
TCP Connect succeeds (three way TCP handshake completes)	Port is open
TCP Connect fails (three way TCP handshake does not complete)	Port is closed

A more complete table would look like:

Packet returned	**Port Status**
SYN/ACK	Port is open
RST from target	Port is closed.
No response	There are 3 possible scenarios, and multiple checks to do in this case: If not on local network, **port is filtered** The host should return either an RST or ICMP packet (see below), or some intervening device should return an ICMP packet. If nothing is returned, a intervening firewall device is simply "swallowing" the packet. If on the local network, check ARP cache. If no arp

	entry exists, **host is down**. If on the local network and arp entry exists, **port is filtered**
RST from other ip address	Port is Filtered
ICMP Port Unreachable ICMP Type 3, Code 3	Port is closed, as outlined in the UDP Port scanning section
All other ICMP Unreachables	Port is Filtered (see UDP Port Scanning Section)

This more complete table has not been implemented at this time, and are being considered for a future release.

UDP Port Scanning

UDP port scanning is significantly more complex than TCP port scanning, especially on the IOS platform. Because there is no three-way handshake, UDP port scanning results must be inferred from other packet types that return when a UDP packet is sent to the port being tested. This means that a method of capturing "interesting packets" that return from a probe must be used. Finally, neither TCL nor the IOS command line has any method of generating a UDP packet.

Several methods were used to overcome these obstacles:

UDP test packets are used by creating IP SLA's to the test port. IP SLA's are generally used to monitor performance of a particular port and/or protocol between two networks, especially if QOS and actual written service level agreements or requirements apply to intervening networks. Care is taken to ensure that SLA control packets are not used, as these "pollute" the output with UDP port 1967 control packets. Using SLA functions involves a configuration change to the routers' running configuration. This means that UDP port scanning using this method should be subject to any change control procedures that govern the hardware platform being used.

Return packets are captured by the router in a two step process. First an access list is created to define what an "interesting packet" might look like – we use access list 111, any ACL name or number might be used if this conflicts with the router configuration. Next, the local log in memory is cleared, and a "debug ip packet 111 detailed" is executed, which will capture the return packets to the log. After a short period of time (3 seconds minimum), the debug is stopped and the access list is removed. This method of packet capture has a few implications. First, debugs can take significant amounts of CPU. On modern hardware, this is normally not appreciable, but should be kept in mind. More importantly, this approach involves both a configuration change and a debug setting, both of which would require a change control request to be approved in most environments.

If used in a consulting engagement, even if change control is not of concern to the client, it might be a good idea to obtain written permission before running UDP scans in this way from an IOS platform.

The following table outlines the various cases that are tested for, and what the resulting port status is inferred to be. As can be seen, the majority of the feedback used to reach a decision is negative or null, it is rare to see actual UDP packets return from a request.

ICMP Port Unreachable packet is returned (ICMP Type 3, Code 3, RFC792)	**Port is considered closed.** The ICMP Port Unreachable response comes from the target host, and indicates that it is not listening on this port
Any other ICMP Destination Unreachable packet is returned These include all ICMP Type 3 packets, with the following codes: 0 Net Unreachable [RFC792] 1 Host Unreachable [RFC792] 2 Protocol Unreachable [RFC792] 4 Fragmentation Needed and Don't Fragment was Set [RFC792] 5 Source Route Failed [RFC792] 6 Destination Network Unknown [RFC1122] 7 Destination Host Unknown [RFC1122] 8 Source Host Isolated [RFC1122] 9 Communication with Destination Network is Administratively Prohibited [RFC1122] 10 Communication with Destination Host is Administratively Prohibited [RFC1122] 11 Destination Network Unreachable for Type of Service [RFC1122] 12 Destination Host Unreachable for Type of Service [RFC1122] 13 Communication Administratively Prohibited [RFC1812] 14 Host Precedence Violation [RFC1812] 15 Precedence cutoff in effect [RFC1812]	**Port is considered "filtered"** These packets are generally returned by network gear between the scanner and target, such as firewalls or routers. These indicate that the intervening gear is blocking the UDP probe packet with an ACL or other Firewall mechanism before it reaches the target host.
UDP Packet returned from the target port	**Port is considered to be "open".** A return packet is a sure sign that the port is answering, however in most cases UDP ports do not return data when probed with zero-data packets.

Nothing is returned	**Port is considered to be "open/filtered".** This is the most frequent return if a port is open. Unfortunately, it is also what is returned on many firewalls if native IDS/IPS features are enabled. For this reason, an "all quiet" situation is generally inconclusive.

Ping Scanning

Ping scanning is very straightforward. Target hosts are sent ICMP Echo Requests (pings), and hosts that return ICMP Echo Replies are considered to be up, and all other hosts are considered to be down. The native Cisco IOS "ping" command is called to execute the echo request.

List Scan

A list scan simply lists the target addresses and ports that would be scanned. It is often used as a "preflight check" on the scan, to ensure that syntax is correct. It can also be used as input for a change control request, should one be required.

Notes on Platform Impact and Change Control

As discussed, TCP scanning has a relatively low but measurable impact on the operational platform – since the "socket" command used in the TCP connect scan is part of the TCL language, no special measures are required to perform this function.

One thing to note in all scans is that IOS will view any scan run as "idle time", as there is no keyboard activity during a scan. Ensure that the vty "exec-timeout" is long enough to accommodate extended scan runs, or they will be simply dropped when the vty session is terminated.

The main operational impact on the platform is memory utilization, which is easily quantified (more on this in the next section).

Similarly, the ping scan simply uses the native "ping" command in IOS in an exec call, so has minimal, quantifiable memory utilization, and requires no configuration changes. The List Scan has almost no impact at all, as it simply prints the scan targets to stdout.

However, UDP port scanning has several specific impacts.

- In order to send UDP packets, the running configuration is modified to create an IP SLA section. This is removed after each port scan is completed.
- Similarly, access list 111 is created to define "interesting" return traffic from a UDP scan, which is a second change to the running configuration.
- The "clear log" command simply doesn't work in cisco's TCL implementation. To clear the log for each run, buffered logging is turned off then back on again – this achieves the exact same goal, but again is a running configuration change.
- All of these will create issues around approval of change control in a well run IT organization. In addition, these configuration changes will generate network alerts on many networks.
- Finally, the use of debugs in capturing the return traffic might also require approval under a change control process.

All of these issues will, on many networks, mean that UDP scanning is not practical with this tool.

Resource Utilization

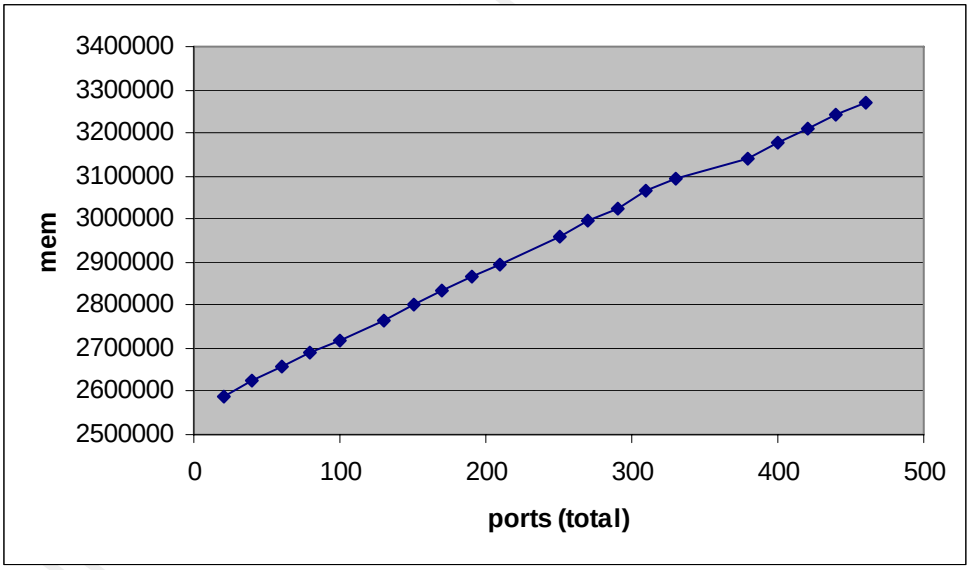
Because routers are such critical devices, when adding new functions it is always important to gauge the impact of these new functions on performance in delivering their core functions. In particular, memory usage and cpu utilization are the two most important factors, both are easily measured. The memory usage of port scanning using TCL was especially interesting.

Memory Utilization

When scanning TCP ports, measuring memory allocation shows a fixed initial memory block used, then an almost straight line increase of memory usage as the total port count increases. Multiple IP's do not contribute significantly to memory utilization, the critical factor is the total number of ports scanned.

	TCP Port Count				
Addresses	20	40	60	80	100
1	2589644	2627088	2656128	2689332	2718372
2	2766588	2802504	2832996	2866200	2895240
3	2960112	2997480	3026520	3063888	3092928
4	3141144	3178512	3207552	3242208	3269796

TCP Port Scanning Memory Utilization



These can be represented closely (less than 0.5% error on each value)

`Memory = (IP's * Ports * 1544) + 2568474`

Or, in more general terms:

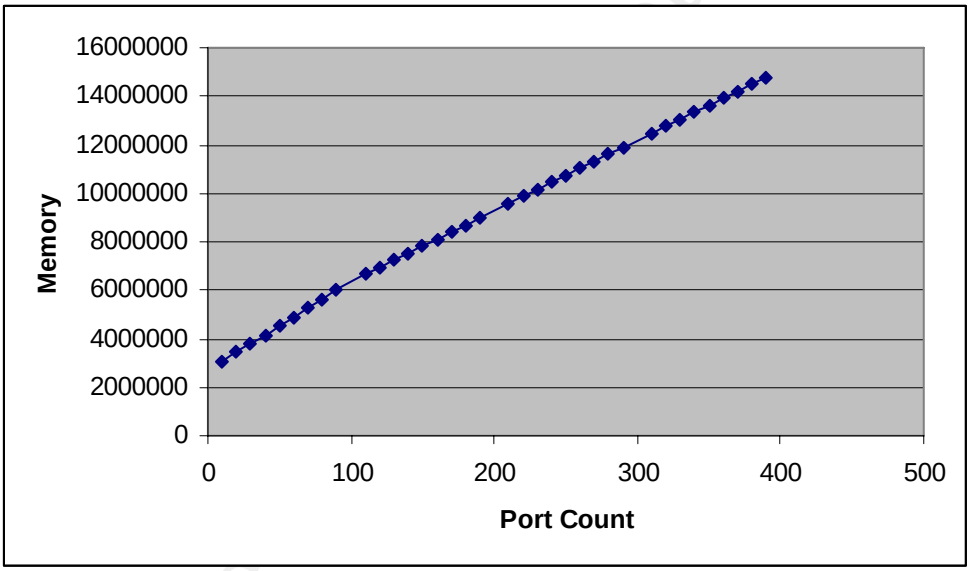
`Memory = (Total TCP Ports) * 1544 + 2568474`

(Correlation Coefficient $R^2 = 0.999$, where $R^2 = 1$ is a perfect fit)

Memory Utilization when scanning UDP ports shows similarly linear behaviour.

	UDP Port Count								
Addresses	10	20	30	40	50	60	70	80	90
1	3021456	3429924	3762796	4141488	4515748	4884844	5259228	5632668	6002124
2	6667996	6962216	7252496	7537384	7827084	8120336	8410392	8695720	8984908
3	9570932	9856068	10145624	10439668	10729500	11014544	11303264	11597992	11892280
4	12470828	12764620	13054804	13339756	13628816	13922740	14207852	14497884	14787640

UDP Port Scanning Memory Utilization



These can be represented closely by:

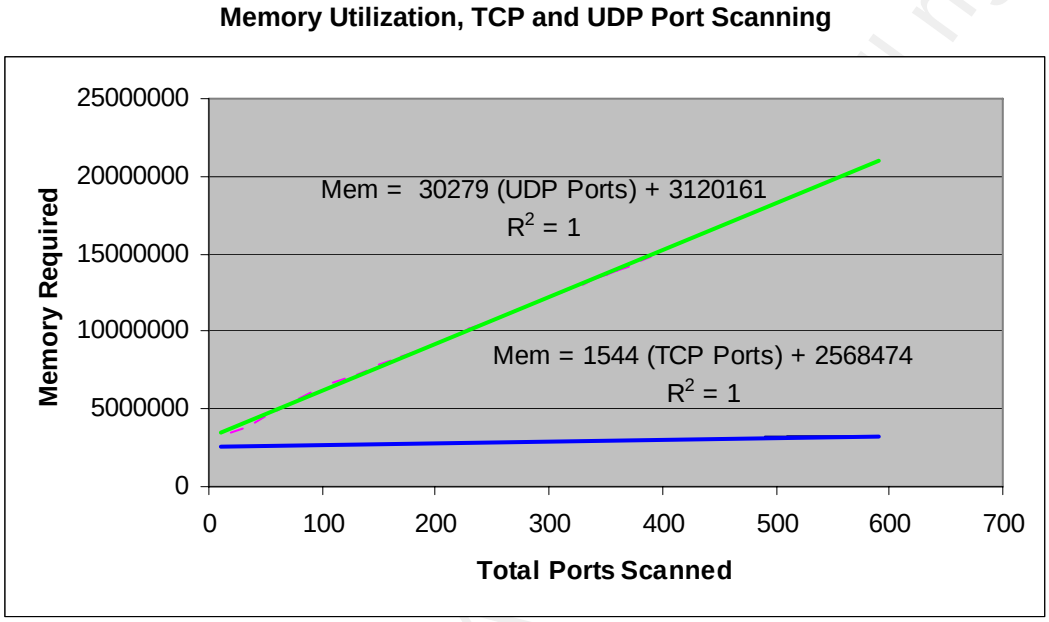
Memory = (IP's * Ports * 30279) + 3120161

Or, in more general terms;

Memory = (Total UDP Ports) * 30279 + 3120161

(Correlation Coefficient $R^2 = 0.999$, where $R^2 = 1$ is a perfect fit)

The thing to note in this, is that the memory required for UDP port scanning is significantly higher than for TCP port scanning. Plotting both functions on the same graph shows this difference dramatically.



On a lightly loaded router with 256MB of DRAM, a UDP port scan of a full class C network for 35 ports will exceed the physical memory on the router. In a more realistic scenario, the **Processor Pool Free Memory** on such a router (3640, IP Plus feature set used as an example) will typically be in the 50MB range. A UDP port scan of a full class C network for 5 ports will exceed this value.

Resource Utilization Watermarks

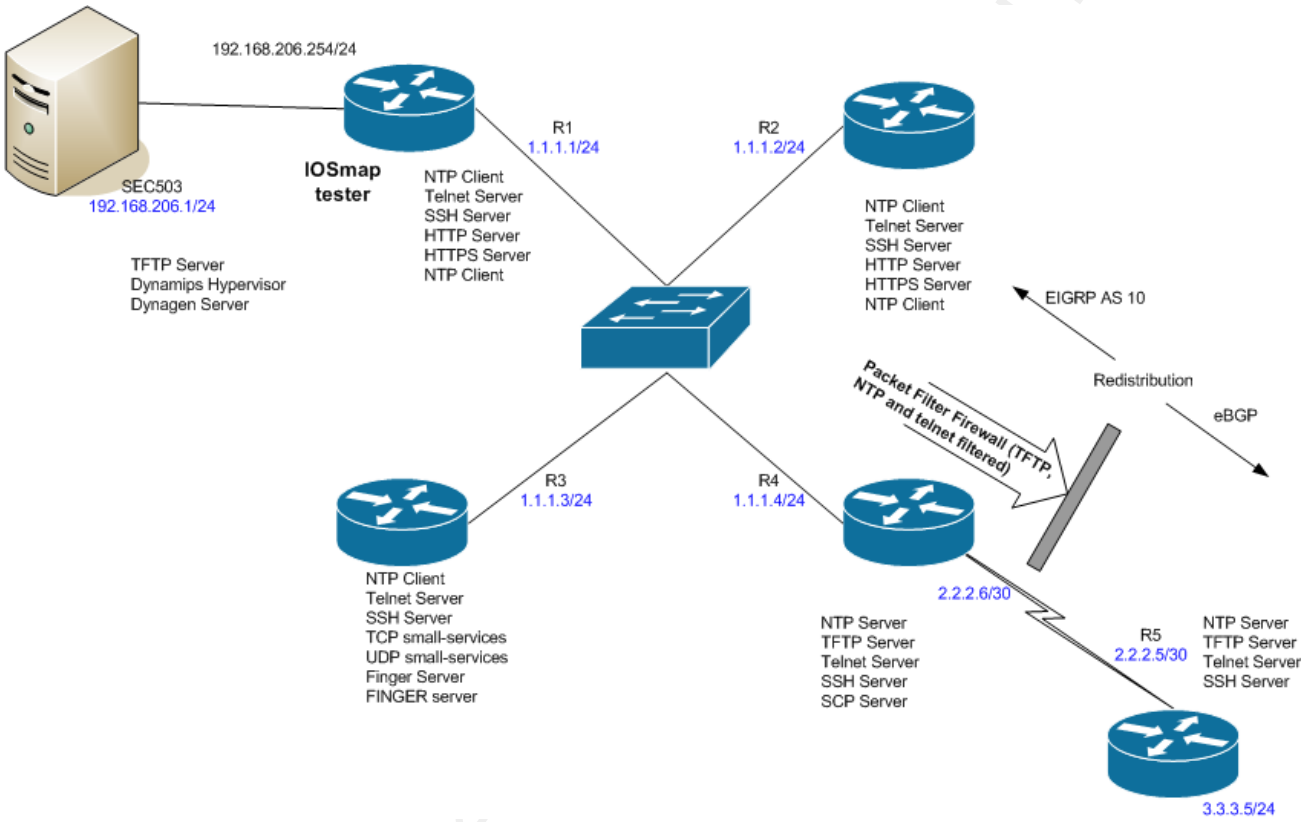
If it is calculated that a given scan will exceed 50% of the available memory, the user is presented with a message and the opportunity to stop the scan. If it is calculated that a given scan will exceed 75% of the available memory, the scan is simply terminated with an error message.

CPU Utilization

CPU Utilization was uniformly less than 5% in all test TCP and UDP scans. This impact would be considered acceptable in most cases.

Running IOSmap (Network Diagram and Examples)

An example network (shown below) was constructed to demonstrate the use of the IOSmap tool:



In the example runs, IOSmap is called from a tftp path. This was done for simplicity, and is not recommended in a production environment for security reasons. Since the tftp protocol does not allow for authentication or encryption, a tftp server can easily be compromised, and TCL scripts stored on it could easily be updated to include rootkits or other "malcode". Using a man-in-the-middle attack, TCL scripts could simply be intercepted and similarly modified in transit between the tftp server and client, without targetting anything on the tftp server at all. In a production situation, an SCP server would generally be recommended, as it provides for both encryption and authentication.

Sample IOSmap runs are shown below:

Scan 1 – a TCP scan of targeted hosts and ports

```
R1#tclsh tftp://sec503/iosmap.tcl 1.1.1.1-5,2.2.2.5 -p7-9,13,19,22-24,80,443
Loading iosmap.tcl from 192.168.206.1 (via FastEthernet0/0): !
[OK - 14830 bytes]

Loading services.list from 192.168.206.1 (via FastEthernet0/0): !
[OK - 42121 bytes]



Starting IOSmap 0.9 ( http://www.defaultroute.ca ) at 2002-03-01 18:18 UTC

Free Memory on Platform = 47298216  / Memory required for this scan =
2661114

Interesting ports on host 1.1.1.1
PORT     STATE      SERVICE
7/tcp   closed     echo
8/tcp   closed     .
9/tcp   closed     discard
13/tcp   closed     daytime
19/tcp   closed     chargen
22/tcp   open       ssh
23/tcp   open       telnet
24/tcp   closed     priv-mail
80/tcp   open       http
443/tcp   open       https



Interesting ports on host 1.1.1.2
PORT     STATE      SERVICE
7/tcp   closed     echo
8/tcp   closed     .
9/tcp   closed     discard
13/tcp   closed     daytime
19/tcp   closed     chargen
22/tcp   open       ssh
23/tcp   open       telnet
24/tcp   closed     priv-mail
80/tcp   open       http
443/tcp   open       https



Interesting ports on host 1.1.1.3
PORT     STATE      SERVICE
7/tcp   open       echo
8/tcp   closed     .
9/tcp   open       discard
13/tcp   open       daytime
19/tcp   open       chargen
22/tcp   open       ssh
23/tcp   open       telnet
24/tcp   closed     priv-mail
80/tcp   open       http
443/tcp   open       https



Interesting ports on host 1.1.1.4
PORT     STATE      SERVICE
```

```
7/tcp   closed    echo
8/tcp   closed    .
9/tcp   closed    discard
13/tcp   closed    daytime
19/tcp   closed    chargen
22/tcp   open      ssh
23/tcp   open      telnet
24/tcp   closed    priv-mail
80/tcp   open      http
443/tcp   open      https



Host 1.1.1.5 is unavailable



Interesting ports on host 2.2.2.5
PORT     STATE      SERVICE
7/tcp   closed    echo
8/tcp   closed    .
9/tcp   closed    discard
13/tcp   closed    daytime
19/tcp   closed    chargen
22/tcp   open      ssh
23/tcp   closed    telnet
24/tcp   closed    priv-mail
80/tcp   open      http
443/tcp   closed    https
```

Scan 2 – a UDP scan of targeted hosts and ports

```
R1#tclsh tftp://sec503/iosmap.tcl 1.1.1.1-5,2.2.2.5 -p7-9,68-69,123 -sU
Loading iosmap.tcl from 192.168.206.1 (via FastEthernet0/0): !
[OK - 14830 bytes]

Loading services.list from 192.168.206.1 (via FastEthernet0/0): !
[OK - 42121 bytes]



Starting IOSmap 0.9 ( http://www.defaultroute.ca ) at 2002-03-01 18:21 UTC

Free Memory on Platform = 47293508  / Memory required for this scan =
4210205

Interesting ports on host 1.1.1.1
PORT     STATE      SERVICE
7/udp   closed    echo
8/udp   closed    .
9/udp   closed    discard
68/udp   closed    dhcpc
69/udp   closed    tftp
123/udp   open    ntp



Interesting ports on host 1.1.1.2
PORT     STATE      SERVICE
7/udp   closed    echo
8/udp   closed    .
9/udp   closed    discard
68/udp   closed    dhcpc
69/udp   open    tftp
123/udp   open    ntp



Interesting ports on host 1.1.1.3
```

```
PORT    STATE     SERVICE
7/udp   open      echo
8/udp   closed    .
9/udp   open      discard
68/udp  closed    dhcpc
69/udp  closed    tftp
123/udp  open     ntp



Interesting ports on host 1.1.1.4
PORT    STATE     SERVICE
7/udp   closed    echo
8/udp   closed    .
9/udp   closed    discard
68/udp  closed    dhcpc
69/udp  open      tftp
123/udp  open     ntp



Host 1.1.1.5 is unavailable



Interesting ports on host 2.2.2.5
PORT    STATE     SERVICE
7/udp   closed    echo
8/udp   closed    .
9/udp   closed    discard
68/udp  closed    dhcpc
69/udp  open      tftp
123/udp  open     ntp
```

References

Brent Welch, Ken Jones (2003). Practical Programming in Tcl and Tk (4th Edition). New Jersey: Prentice Hall PTR, 2003.

"TCL Reference Manual", 25 June, 2008 <http://tmml.sourceforge.net/doc/tcl/>.

"Cisco IOS Scripting with Tcl", 25 June, 2008. <http://www.cisco.com/en/US/docs/ios/12_3t/12_3t2/feature/guide/gt_tcl.html>.

NMAP Reference Guide, 25 June, 2008 < http://nmap.org/book/man.html >.

"A Little CIDR Calculator", 05 May, 2008. <http://wiki.tcl.tk/8909>.

"RFC 792 - INTERNET CONTROL MESSAGE PROTOCOL" Sept 1981. <http://www.ietf.org/rfc/rfc0792.txt>.

"RFC 768 – USER DATAGRAM PROTOCOL" 28 Aug 1980. <http://tools.ietf.org/html/rfc768>.

"RFC 761 – TRANSMISSION CONTROL PROTOCOL", Jan 1980. <http://tools.ietf.org/html/rfc761>.

Appendix: Full Source Code Listing, Commented

# set defaults if not over-ridden at cmd line	
set pingit 1 ; # ping scan set to no (should be yes)	Set default values on variables used throughout
set scantype T ; # default scan is TCP	
set ports ""	
set portslist 0	
set timeout 1	
set timeoutms 500	
set waittime 4000	
set svcarraymax 4096	
set fullpath $argv0	This is the full pathspec of the called script (IOSmap.tcl)
set reasoncode 0 set reason "NULL"	
	This application consists of many procedures (procs)
	These are at the beginning of the listing, the main executable
	Is at the bottom of this listing
proc setloadpath { fullpath } {	Get the path that IOSmap was called from
global loadpath	We'll need that later in order to
set firstchar 0	
set endchar [expr [string last "/" $fullpath] -1]	
set loadpath [string range $fullpath $firstchar $endchar]	
return $loadpath	
}	
proc syntaxhelp {} {	This helptext subroutine prints all the cmd line sytax available
puts stdout "\7========== ========"	\7 prints an ASCII 7 character (aka "bell"). This emits a "beep" when printed to STDOUT
puts stdout "IOScan 0.1"	
puts stdout " Usage: IOScan <Scan Type> <Options> <target specifications>"	
puts stdout "HOST DISCOVERY:"	
puts stdout " -P0 Treat all hosts as online - skip Ping test"	
puts stdout " -SL List hosts and ports to scan"	
puts stdout "SCAN TYPE:"	
puts stdout " -sP Ping scan only <ICMP ECHO>"	
puts stdout " -sT TCP Connect Scan"	
puts stdout " -sU UDP Scan"	
puts stdout "PORT SPECIFICATION:"	
puts stdout " -p <port ranges> Specify ports to scan. "	
puts stdout " -p22 Scan port 22"	
puts stdout " -p22,23,135-139,445 Scan ports 22, 23, 135, 136, 137, 138, 139, 445"	
puts stdout "TARGET SPECIFICATION:"	
puts stdout " CIDR, IP range and single IPs are all a supported - comma delimited"	
puts stdout " For example:"	
puts stdout " 192.168.10.0/24,192.168.17.21-34,192.168.40.1"	
}	
proc memcalc { scantype } {	This procedure calculates the estimated memory usage the scan run will take.
global iplist	The iplist is the list of all ip's to be scanned

global portlist	The portlist is the list of ports to be scanned
if { $scantype == "T" } {	If the scan is for TCP ports, define the equation characteristics
set gradient 1544	
set intercept 2568474	
} else {	Similarly, these are the characteristics for a UDP scan.
set gradient 30279	
set intercept 3120161	
}	
set factor1 50 ; # watermark to ask for a y/n to proceed	These are our safety factors (50 and 75%)
set factor2 75 ; # watermark to force an exit	
set ipcount [llength $iplist]	Get the total count of ip's to scan
set portcount [llength $portlist]	Get the total count of ports to scan
set calcmem [expr ($portcount * $ipcount * $gradient)]	Calculate the total memory to be consumed
set calcmem [expr ($calcmem + $intercept)]	
set i [exec "sho proc mem | i Processor Pool"]	Get the free memory available in the Processor Memory Pool
set memfree [lindex $i [expr ([llength $i] - 1)]]	
set memlimit1 [expr ($memfree / 100 * $factor1)]	Memory limit 1 (50% of free memory)
set memlimit2 [expr ($memfree / 100 * $factor2)]	Memory limit 2 (75% of free memory)
puts stdout "Free Memory on Platform = $memfree / Memory required for this scan = $calcmem"	Print memory values – this keeps it top-of-mind for anyone running the script
puts stdout " "	
if { $calcmem > $memlimit2 } {	We're over 75% - exit the application
puts stdout "\7The resources estimated for your scan will exceed $factor2\%"	
puts stdout "of your available memory total of $calcmem"	
puts stdout "Execution cannot proceed without impacting primary device functions"	
return 1	
} elseif { $calcmem < 0 } {	Our calc has overflowed the precision of TCL, so we know that it's more than physical ram of any router platform
puts stdout "\7The resources used by your scan will exceed the physical memory installed"	
puts stdout "on your platform. Execution cannot proceed without impacting"	
puts stdout "primary device functions"	
return 1	
} elseif { $calcmem > $memlimit1 } {	Memory usage calc is over 50% - ask for confirmation to proceed.
puts stdout "\7The resources used by your scan will exceed $factor1%"	
puts stdout "of your available memory total of $calcmem"	
puts stdout "This may impact primary device functions"	
puts -nonewline stdout "do you wish to proceed (y/n) ==> "	
flush stdout	
set response [gets stdin];	
if { $response == "y" } { return 0 } else { return 1 }	
} elseif { $calcmem < $memlimit1 } { return 0 }	Final case, all is well, proceed with the scan.
}	
proc IPtoHex { IP } {	Convert an ip address to it's hexadecimal equivalent
binary scan [binary format c4 [split $IP .]] H8 Hex	
return $Hex	
}	
proc hex2dec {hexvalue} {	Convert a hexadecimal value to decimal
set decvalue [format "%u" [expr 0x$hexvalue]]	
return $decvalue	
}	
proc dec2hex { decvalue } {	Convert decimal value to hexadecimal
set hexvalue [format "%#010X" [expr $decvalue]]	

return $hexvalue	
}	
proc Hex2IP { Hex } {	Convert a hexadecimal value to it's equivalent IP address
# first trim off leading "0x" if it's there	
if { [string length $Hex] == 10 } { set Hex [string range $Hex 2 9] }	
binary scan [binary format H8 $Hex] c4 IPtmp	
foreach num $IPtmp {	
lappend IP [expr ($num + 0x100) % 0x100]	
}	
set IP [join $IP .]	
return $IP	
}	
proc isipvalid { IP } {	Is the ip address valid?
# only digits'n'dots	
regsub -all {[.0-9]} $IP {} scratchvar	Are all the chars either numeric or periods?
if { $scratchvar != "" } {	
return 0	
}	
# 4 octets means exactly 3 dots	
regsub -all {[0-9]} $IP {} scratchvar	Do we have exactly 3 periods?
if { $scratchvar != "..." } {	
return 0	
}	
# is each octet betw 0 and 255?	Are all values between 0 and 255?
foreach b [split $IP .] {	
if { [string length $b] == 0 } {	
return 0	
}	
set ob $b	
#parse out leading zeros	
scan $b %d b	
if { $b < 0 | $b > 255 } {	
return 0	
}	
}	
return 1	Final case, all is well
}	
proc iscidrvalid { CIDR } {	Is the network subnet mask (in CIDR notation) valid?
# numeric check	
regsub -all {[0-9]} $CIDR {} scratchvar	
if { [string length $scratchvar] != 0 } {	
return 0	
}	
#convert to numeric, check values	
#because this is running on a router, mask <8 is not acceptable due to scan time.	Values under 8 are too many IP's to scan on a router
# mask of /31 or /32 is also not acceptable	Values over 30 mess up the math, and are better specified as a range or discrete IP's anyway
scan $CIDR %d CIDR	
if { $CIDR < 8 | $CIDR > 30 } {	
return 0	

}	
return 1	Final case, all is well, proceed.
}	
proc ipCIDR { net } {	Parse out a network specified as a CIDR block into discrete IP's
global iplist	
set work1 [split $net /]	
set ip1 [lindex $work1 0]	
if { ! [isipvalid $ip1] } {	
puts stdout "\7Invalid IP address specified ==> $ip1"	
puts " "	
return 1	
}	
scan $net {%d.%d.%d.%d/%d} a b c d bits	
if { ! [iscidrvalid $bits] } {	
puts stdout "Invalid Netmask address specified ==> /$bits"	
puts stdout "Because of platform considerations, subnet mask must be >=8 or <=30"	
puts " "	
return 1	
}	
set hexmask [expr {0xffffffff & (0xffffffff << (32-$bits))}]	Get the broadcast ip address
set bnet [hex2dec [IPtoHex $ip1]]	
set realnet [expr $bnet & $hexmask]	
set firstip [expr $realnet+1]	The first ip is the network + 1
set bcast [expr $bnet | ($hexmask ^ 0xffffffff)]	
set lastip [expr $bcast - 1]	The last ip to scan is the broadcast – 1
for { set j $firstip } { $j <= $lastip } { incr j} {	Now, loop from first to last IP, and populate the IP list
set work1 [dec2hex $j]	
lappend iplist [Hex2IP $work1]	
}	
return 0	
}	
proc iprange { net } {	Parse out a network ,specified as with a range in octet 4
global iplist	
set work1 [split $net -]	
set ip1 [lindex $work1 0]	Get the first ip
set maxoct4 [lindex $work1 1]	Get the last ip
if { ! [isipvalid $ip1] } {	Are both ip's valid?
puts stdout "Invalid IP address specified ==> $ip1"	
return 1	
}	
scan $ip1 {%d.%d.%d.%d} a b c d	
set ipmax $a.$b.$c.$maxoct4	
if { ! [isipvalid $ipmax] } {	
puts stdout "Invalid IP address specified ==> $ipmax"	
return 1	

}	
if { $d > $maxoct4 } {	Is the range specified low-to-high?
puts stdout "Invalid IP address range specified ==> $ip1-$maxoct4"	
return 1	
}	
for { set j $d} {$j <= $maxoct4 } { incr j} {	
lappend iplist $a.$b.$c.$j	
}	
return 0	Final case, all is well
}	
proc parsenet { networklist } {	Parse the complete IP list out from the cmd line string
global iplist	
set netlist [split $networklist ,]	Split out the commas
foreach net $netlist {	
if { [string first / $net] >0 } {	Is it a CIDR block?
set retval [ipCIDR $net]	
} elseif { [string first - $net] >0} {	Is it specified as a range?
set retval [iprange $net]	
} else {	
if { ! [isipvalid $net] } {	Is it a single ip address?
puts stdout "Invalid IP address specified ==> $net"	
return 1	
}	
lappend iplist $net }	
}	
return 0	Final case, all is well
}	
proc pinger {ip timeout} {	Ping a host and tell me if it exists
set pingretry 3	
# returns a 1 if any icmp echo replies make it back, otherwise returns a 0	
if { [regexp "(!)" [exec "ping $ip timeout $timeout repeat $pingretry"]] } { return 1 } else { return 0 }	
}	
proc scantcpconnect {host port} {	TCP Connect scan of a discrete ip address and port
global timeout	
global reason	
set timeout1 [expr $timeout*1000]	Convert the timeout to milliseconds
catch { socket $host $port } sock	Connect
after $timeout1	Wait for the timeout
if { [string first sock $sock] == 0} {	If the string “sock” is returned from the socket command, the port is open
catch { close $sock }	
return "open "	
} else { return "closed" }	If not, it's closed
}	
proc udpscan { ip port } {	Attempt to see if a UDP port is open on a single ip address
# timers should be global, logfile should NOT be global	
global timeoutms	These timeouts are hard-coded for now.
global waittime	
global reason	
ios_config "no logging buffer"	Clear the buffered log (this is a config change)

ios_config "logging buff 8192 debug"	
set retcode "error" ; # just in case, give retcode a value	
# set up the list of interesting packets to look for (ie set up packet capture filter)	Create the access list that identifies what we're looking for
ios_config "access-list 111 permit udp any host $ip eq $port"	
ios_config "access-list 111 permit udp host $ip eq $port any"	
ios_config "access-list 111 permit icmp host $ip any unreach"	
# now, watch for these packets (ie start your packet capture)	
exec "debug ip packet 111 det"	Log occurrences of matches to our ACL to the log (ie – capture packets)
# next, send test udp packets to trigger responses	
ios_config "ip sla monitor 111" "type udpEcho dest-ipaddr $ip dest-port $port control disable" "time $timeoutms" "freq 1"	Now, lets create an IP SLA to send some udp probe packets Note that control packets are DISABLED
ios_config "ip sla mon schedule 111 life forever start now"	Schedule the IP SLA to run
after $waittime ; # wait - 2sec is generally enough for the log to catch up	
# now clean up confg and debug changes	Clean up:
exec "no debug ip pack 111 det"	• Stop the packet capture
ios_config "no access-list 111"	• Clear the ACL
ios_config "no ip sla monitor 111"	• Erase the IP SLA
set startpos "dst=$port"	
set logfile [exec "show log"]	Move the log into a variable list
set ipstart 0	
set portunreach 0	
set unreach 0	
# first, find the first occurrence of our target in the log	
set ipstart [string first $startpos $logfile]	Where is the first occurrence of a sent packet?
#now, look for icmp type 3, or icmp type 3 code 3, occuring after this ip value	Look for ICMP Port unreachable replies returned *after* this packet
# (ie - make sure we're not reading a previous status).	
if { $ipstart > 0 } {	
Set unreach [string last "ICMP type=3" $logfile]	Find the last ICMP UNREACHABLE
Set portunreach [string last "ICMP type=3, code=3" $logfile]	Find the last ICMP PORT UNREACABLE
set udpreturn [string last "UDP src=$port" $logfile]	Find the last UDP port from the target
set retcode "open/filtered" ; # set the case for no packets back at all	The default case is no packets back – open/filtered
if { $unreach > $ipstart } { set retcode "filtered" }	ICMP unreachable indicates filtered
if { $portunreach > $ipstart } { set retcode "closed" }	Is it closed? (ICMP port unreachable) – overwrites "filtered" case above as it's a more specific ICMP unreachable
if { $udpreturn > $ipstart } { set retcode "open" }	Return packet indicates open port
} else { set retcode "open/filtered" } ; # this accounts for no packets back on empty logfile	This case is for an empty logfile (should never occur)
return $retcode	
}	
proc scanit {localportlist localnetworklist scantype pingit} {	Generic scan a network list with a portlist, all scan types
global timeout	
global svctcp	

global svcudp	
global svcarraymax	
global reason	
global reasoncode	
foreach host $localnetworklist {	For each host in the list ….
# set existance default in case -P0 (no ping) is specified	
set hostexist 1	If we're not pinging the hosts, tell me that they are all up.
if {$pingit == 1} { set hostexist [pinger $host $timeout] }	If we're pinging, do it
if { $scantype == "P" } {	Is it a ping scan?
if { $hostexist ==1 } {	If so, simply print the results and go on to the next
puts stdout "Host $host is up"	
} else { puts "Host $host is down" }	
} else {	
if {$hostexist == 1 } {	Is the host up?
puts stdout "Interesting ports on host $host"	
puts -nonewline stdout "PORT STATE SERVICE"	Print the port title line – REASON is only printed if requested
if {$reasoncode == 1} {puts -nonewline stdout " REASON"}	
puts ""	
foreach port $localportlist {	
if { $scantype == "T" } {	Is it a TCP scan?
set state [scantcpconnect $host $port]	If so, proceed
set proto "tcp"	
if {$port <= $svcarraymax} {	
set service $svctcp($port)	Format the results
}	
} elseif {$scantype == "U" } {	Is it a UDP scan?
set state [udpscan $host $port]	If so, proceed
set proto "udp"	
if {$port <= $svcarraymax} {	Format the results
set service $svcudp($port)	
}	
} elseif {$scantype == "L" } {	Is it a list scan?
set proto "tcp"	If so, format the results (assume TCP)
set state "unscanned"	
if {$port <= $svcarraymax} {	
set service $svctcp($port)	
}	
}	
puts -nonewline stdout "$port/$proto $state $service"	Print the results for this ip and port
if {$reasoncode == 1} { puts -nonewline stdout " $reason"}	Again, the reason code is only printed if requested
puts stdout ""	
	On to the next port
}	
} else { puts stdout "Host $host is unavailable" }	Host is not up
puts stdout "\n\n"	
}	On to the next ip
}	
return	
}	
proc parseports { ports } {	Parse the ports out from the command line string
global portlist	
set localportlist [split $ports ,]	

foreach port $localportlist {	
if {[string first - $port] > 0} {	Is this a port range?
set localplist [split $port -]	
for {set lport [lindex $localplist 0]} {$lport <= [lindex $localplist 1]} {incr lport} {	Loop through
if {$lport > 0 && $lport <65535 } {	Ensure that we are >0 and < 64K
lappend portlist $lport	
} else {	
puts stdout "Invalid port value ==> $lport"	Invalid port error
return 1	
}	
}	
} else {	
if {$port >0 && $port <65535 } {	This is a single port - again, check
lappend portlist $port	
} else {	
puts stdout "Invalid port value ==> $port"	
return 1	
}	
}	
}	
return 0	
}	
proc getservices { loadpath } {	Get the services file (we have names for the same services as NMAP does
global svctcp	
global svcudp	
global svcarraymax	
for {set i 1} {$i < $svcarraymax} {incr i} {	Populate the array with dots first (undefined ports)
set svctcp($i) "."	
set svcudp($i) "."	
}	
set svcfile "/services.list"	
if [catch {open $loadpath$svcfile r} fileId] {	Now, get the services file from the same location we loaded IOSmap from.
puts stderr "Cannot open services file"	Error in file read
} else {	
set services [read $fileId]	Read the file into a temp list
close $fileId	
}	
foreach record $services {	For each line in the file
set localrec [split $record ,]	
set localproto [lindex $localrec 0]	
set localport [lindex $localrec 1]	
set localsvc [lindex $localrec 2]	
switch $localproto {	
tcp {set svctcp($localport) $localsvc }	Populate the tcp and udp service description arrays
udp {set svcudp($localport) $localsvc }	
}	
}	
return	
}	

	Main Script Execution Starts Here
#process cmd line arguments	First, lets get the cmd line arguments
foreach arg $argv {	
switch -glob -- $arg {	Depending on the cmd line switches
-sU {set scantype U}	Populate the appropriate variables
-sT {set scantype T}	
-sP {set scantype P ; set ports 1}	
-sL {set scantype L ; set ports 1 ; set pingit 0}	
-P0 {set pingit 0}	
-p* {set ports $arg}	
-h { set scantype "H" }	
default {set network $arg}	If there is no switch, assume that it's a network value
}	
}	
if {$scantype != "P"} {	Is it NOT a Ping sweep?
set loadpath [setloadpath $fullpath]	(Services file is not required for pings)
#populate the arrays defining the tcp and udp service descriptions	
#depends on a data file in a hard-coded directory	
getservices $loadpath	
}	
# dump out intro line	
puts stdout "\n\n"	
puts stdout [clock format [clock seconds] -format {Starting IOSmap 0.9 (http://www.defaultroute.ca) at %Y-%m-%d %H:%M %Z}]	Print out the "splash" line Hopefully this will motivate some to set time (either static or via NTP) and timezone on their gear
puts ""	
if {$scantype != "H" } {	Is it NOT a request for syntax help?
# trim "-p out of ports arg, parse out the ports to a list of discrete values	
set ports [string trimleft $ports -p]	Pull the "-p" off the port list
set ok1 [parseports $ports]	Now, get the port list
# parse network values out to a discrete list of ip addresses	
set ok2 [parsenet $network]	Get the network list
set ok [expr $ok1+$ok2]	
if { $ok == 0 } {	Are the port list and network list both ok?
set retcode [memcalc $scantype]	Calculate the memory utilization
if {$retcode == 0 } {	Is the memory situation ok?
# scan the list of ports and ip's as specified	
scanit $portlist $iplist $scantype $pingit	If so, scan the ports
}	
} else {	
syntaxhelp	Print the syntax help text
}	
}	