Global Information Assurance Certification Paper

Edward Fuller

Netcat: Network Kitty or Black (Hat) Panther

Netcat is a Unix program that transfers information across a network via the TCP or UDP protocols. 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 network debugging and exploration tool, which can accommodate nearly any kind of connection you desire. The actual program is named (nc). It also has more sinister potential than is immediately obvious.

I will first describe the program and then reveal it's darker capabilities.

Netcat

"Nc host port" creates a TCP connection to the given port on the given target host. The standard input is then sent to the host, and anything that comes back across the connection is sent to your standard output. This continues indefinitely, until the network side of the connection shuts down. Note that this behavior is different from most other applications which shut everything down and exit after an end-of-file on the standard input.

Netcat can also function as a server, by listening for inbound connections on arbitrary ports and then doing the same reading and writing. With minor limitations, netcat doesn't really care if it runs in "client" or "server" mode -- it still shovels data back and forth until there isn't any more left. In either mode, shutdown can be forced after a configurable time of inactivity on the network side.

And it can do this via UDP too, so netcat is possibly the "udp telnet-like" application you always wanted for testing your UDP-mode servers. UDP, as the "U" implies, gives less reliable data transmission than TCP connections and some systems may have trouble sending large amounts of data that way, but it's still a useful capability to have.

Some advantages of nc over telnet. Telnet has the "standard input EOF" problem, so one must introduce calculated delays in driving scripts to allow network output to finish. This is the main reason netcat stays running until the *network* side closes. Telnet also will not transfer arbitrary binary data, because certain characters are interpreted as telnet options and are thus removed from the data stream. Telnet also emits some of its diagnostic messages to standard output, where netcat keeps such things religiously separated from its *output* and will never modify any of the real data in transit unless you *really* want it to. And of course telnet is incapable of listening for inbound connections, or using UDP instead. Netcat doesn't have any of these limitations, is much smaller and faster than telnet, and has many other advantages.

Some of netcat's major features are:

- Outbound or inbound connections, TCP or UDP, to or from any ports

- DNS forward/reverse checking, with appropriate warnings
- Can use any local source port
- Can use any locally-configured network source address
- Inherent port-scanning capabilities, with randomizer
- Inherent loose source-routing capability
- Reads command line arguments from standard input
- Slow-send mode, one line every N seconds
- Hex dump of transmitted and received data
- Optional ability to let another program service established connections
- Optional telnet-options responder

Notable functions of Netcat

- Nc provides a basic and easily-modified template for writing other network applications.
- Netcat is at the same time so simple and versatile.
- If no command arguments are given at all, netcat asks for them, reads a line from standard input, and breaks it up into arguments internally.
- The host argument can be a name or IP address.
- A port argument is required for outbound connections, and can be numeric or a name as listed in /etc/services
- The -v switch controls the verbosity level of messages sent to standard error
- Note that -w also sets the network inactivity timeout.
- UDP connections are opened instead of TCP when -u is specified.
- To obtain a hex dump file of the data sent either way, use "-o logfile".
- Netcat can bind to any local port, subject to privilege restrictions and ports that are already in use. It is also possible to use a specific local network source address if it is that of a network interface on your machine.
- Listen mode will cause netcat to wait for an inbound connection, and then the same data transfer happens.

If netcat is compiled with -DGAPING_SECURITY_HOLE, the -e argument specifies a program to exec after making or receiving a successful connection. In the listening mode, this works similarly to "inetd" but only for a single instance. Use with GREAT CARE. This piece of the code is normally not enabled; if you know what you're doing, have fun. This hack also works in UDP mode. Note that you can only supply -e with the name of the program, but no arguments. If you want to launch something with an argument list, write a two-line wrapper script or just use inetd like always.

If netcat is compiled with -DTELNET, the -t argument enables it to respond to telnet option negotiation [always in the negative, i.e. DONT or WONT]. This allows it to connect to a telnetd and get past the initial negotiation far enough to get a login prompt from the server. Since this feature has the potential to modify the data stream, it is not enabled by default. You have to understand why you might need this and turn on the #define yourself.

Port-scanning is a popular method for exploring what's out there. Netcat accepts its commands with options first, then the target host, and everything thereafter is interpreted as port names or numbers, or ranges of ports in M-N syntax.

Testing network connectivity using IP source routing, even if it's only to make sure the firewalls are blocking source-routed packets. On systems that support it, the -g switch can be used multiple times [up to 8] to construct a loose-source-routed path for your connection, and the -G argument positions the "hop pointer" within the list. If your network allows source-routed traffic in and out, you can test connectivity to your own services via remote points in the internet. Note that although newer BSD-flavor telnets also have source-routing capability, it isn't clearly documented and the command syntax is somewhat clumsy. Netcat's handling of "-g" is modeled after "traceroute".

Netcat tries its best to behave just like "cat". It currently does nothing to terminal input modes, and does no end-of-line conversion. Standard input from a terminal is read line by line with normal editing characters in effect. You can freely suspend out of an interactive connection and resume. ^C or whatever your interrupt character is will make netcat close the network connection and exit.

Netcat doubles as a teaching tool, one can learn a great deal about more complex network protocols by trying to simulate them through raw connections.

Netcat is an obvious replacement for telnet as a tool for talking to daemons. For example, it is easier to type "nc host 25", talk to someone's mailer, and just ^C out than having to type ^]c or QUIT as telnet would require you to do. You can quickly catalog the services on your network by telling netcat to connect to well-known services and collect greetings, or at least scan for open ports.

A scanning example: "echo QUIT | nc -v -w 5 target 20-250 500-600 5990-7000" will inform you about a target's various well-known TCP servers, including r-services, X, IRC, and maybe a few you didn't expect. Sending in QUIT and using the timeout will almost guarantee that you see some kind of greeting or error from each service, which usually indicates what it is and what version. [Beware of the "chargen" port, though...]

SATAN uses exactly this technique to collect host information, and indeed some of the ideas herein were taken from the SATAN backend tools. If you script this up to try every host in your subnet space and just let it run, you will not only see all the services, you'll find out about hosts that aren't correctly listed in your DNS. Then you can compare new snapshots against old snapshots to see changes. For going after particular services, a more intrusive example is in scripts/probe.

Netcat can be used as a simple data transfer agent, and it doesn't really matter which end is the listener and which end is the client, input at one side arrives at the other side as output.

Netcat as well can make an outbound connection and then run a program or script on the originating end, with input and output connected to the same network port.
You can use netcat to generate huge amounts of useless network data for various performance testing.

Binding to an arbitrary local port allows you to simulate things like r-service clients, if you are root locally.

Netcat also provides several ways for you to test your own packet filters.

You can use netcat to protect your own workstation's X server against outside access.

Hacking Uses

Now that nc has been described in depth, it's potential as a hacking utility must be recognized. You can use netcat to attack or defend.

The first obvious thing is scanning someone *else's* network for vulnerable services. Files containing preconstructed data, be it exploratory or exploitive, can be fed in as standard input, including command-line arguments to netcat itself to keep "ps" ignorant of your doings. The more random the scanning, the less likelihood of detection by humans, scan-detectors, or dynamic filtering, and with -i you'll wait longer but avoid loading down the target's network.

If you build netcat with GAPING_SECURITY_HOLE defined, you can use it as an "inetd" substitute to test experimental network servers that would otherwise run under "inetd". A script or program will have its input and output hooked to the network the same way. Given that most network services do not bind to a particular local address, whether they are under "inetd" or not, it is possible for netcat avoid the "address already in use" error by binding to a specific address. This lets you [as root, for low ports] place netcat "in the way" of a standard service, since inbound connections are generally sent to such specifically-bound listeners first and fall back to the ones bound to "any". This allows for a one-off experimental simulation of some service, without having to mess around with inetd.conf.

Some configurations of packet filters attempt to solve the FTP-data problem by just allowing such connections from the outside. These come FROM port 20, TO high TCP ports inside -- if you locally bind to port 20, you may find yourself able to bypass

filtering in some cases. Maybe not to low ports "inside", but perhaps to TCP NFS servers, X servers, Prospero, ciscos that listen on 200x and 400x. Similar bypassing may be possible for UDP [and maybe TCP too] if a connection comes from port 53; a filter may assume it's a nameserver response.

Using -e in conjunction with binding to a specific address can enable "server takeover" by getting in ahead of the real ones, whereupon you can snarf data sent in and feed your own back out. At the very least you can log a hex dump of someone else's session. If you are root, you can certainly use -s and -e to run various hacked daemons without having to touch inetd.conf or the real daemons themselves. You may not always have the root access to deal with low ports, but what if you are on a machine that also happens to be an NFS server? You might be able to collect some interesting things from port 2049, including local file handles. There are several other servers that run on high ports that are likely candidates for takeover, including many of the RPC services on some platforms [yppasswdd]. Kerberos tickets, X cookies, and IRC traffic also come to mind. RADIUS-based terminal servers connect incoming users to shell-account machines on a high port, usually 1645. SOCKS servers run on 1080. Do "netstat -a" and get creative.

Using -e to start a remote backdoor shell is another possibility. Easier than constructing a file for inetd to listen on, and you can access-control it against other people by specifying a client host and port. Experience with this truly demonstrates how fragile the barrier between being "logged in" or not really is. If you're already behind a firewall, it may be easier to make an *outbound* connection and then run a shell; a small wrapper script can periodically try connecting to a known place and port, you can later listen there until the inbound connection arrives, and there's your shell. Running a shell via UDP has several interesting features, although be aware that once "connected", the UDP stub sockets tend to show up in "netstat" just like TCP connections and may not be quite as subtle as you wanted. Packets may also be lost, so use TCP if you need reliable connections. But since UDP is connectionless, a hookup of this sort will stick around almost forever, even if you ^C out of netcat or do a reboot on your side, and you only need to remember the ports you used on both ends to reestablish. And outbound UDP-plus-exec connection creates the connected socket and starts the program immediately. On a listening UDP connection, the socket is created once a first packet is received. In either case, though, such a "connection" has the interesting side effect that only your client-side IP address and source port will thereafter be able to talk to it. Instant access control! A non-local third party would have to do all of the following to take over such a session:

- forge UDP with your source address
- guess the port numbers of BOTH ends, or sniff the wire for them
- arrange to block ICMP or UDP return traffic between it and your real
- source, so the session doesn't die with a network write error

Netcat can prevent inadvertently sending extra information over a telnet connection. Use "nc -t" in place of telnet, and daemons that try to ask for things like USER and TERM environment variables will get no useful answers, as they otherwise would from a more

recent telnet program. Some telnetds actually try to collect this stuff and then plug the USER variable into "login" so that the caller is then just asked for a password! This mechanism could cause a login attempt as YOUR real username to be logged over there if you use a Borman-based telnet instead of "nc -t".

Got an unused network interface configured in your kernel [e.g. SLIP], or support for alias addresses? Ifconfig one to be any address you like, and bind to it with -s to enable all sorts of shenanigans with bogus source addresses. The interface probably has to be UP before this works; some SLIP versions need a far-end address before this is true. Hammering on UDP services is then a no-brainer. What you can do to an unfiltered syslog daemon should be fairly obvious; trimming the conf file can help protect against it. Many routers out there still blindly believe what they receive via RIP and other routing protocols. Although most UDP echo and chargen servers check if an incoming packet was sent from *another* "internal" UDP server, there are many that still do not, any two of which [or many, for that matter] could keep each other entertained for hours at the expense of bandwidth. And you can always make someone wonder why she's being probed by nsa.gov.

TCP spoofing possibilities are mostly limited to destinations you can source-route to while locally bound to your phony address. Many sites block source-routed packets these days for precisely this reason. If your kernel does oddball things when sending source-routed packets, try moving the pointer around with -G. You may also have to fiddle with the routing on your own machine before you start receiving packets back. Warning: some machines still send out traffic using the source address of the outbound interface, regardless of your binding, especially in the case of localhost. Check first. If you can open a connection but then get no data back from it, the target host is probably killing the IP options on its end [this is an option inside TCP wrappers and several other packages], which happens after the 3-way handshake is completed.

SYN bombing [sometimes called "hosing"] can disable many TCP servers, and if you hit one often enough, you can keep it unreachable for days.

Does your shell-account provider allow personal Web pages, but not CGI scripts? You can have netcat listen on a particular port to execute a program or script of your choosing, and then just point to the port with a URL in your homepage. The listener could even exist on a completely different machine, avoiding the potential ire of the homepage-host administrators. Since the script will get the raw browser query as input it won't look like a typical CGI script, and since it's running under your UID you need to write it carefully. You may want to write a netcat-based script as a wrapper that reads a query and sets up environment variables for a regular CGI script. The possibilities for using netcat and scripts to handle Web stuff are almost endless.

Oh yea, and one last thing about netcat, now there’s an encrypted version called “Cryptcat”. So sniffing it’s traffic is irrelevant.

In conclusion, Netcat/nc is a versatile tool that can be employed for legitimate or elicit purposes. This is true of any power and should be realized by any security professional

or system administrator that may encounter such things. Be aware, curious, and educated.

With this new sense of paranoia, one must ask, “Is this running on my server?” Maybe it is. Scan yourself and try to telnet to any running ports that you don’t recognize. You never know unless you try.

Sources:

1. The Hobbit, “Netcat”, www.attrition.org, Network Utility, 1998.
2. Cryptcat, URL: www.technotronic.com, Network Utility, 1/30/2001
3. [SECURITY-BASICS] NT "security" (netcat archives list)
   URL: www.securityfocus.com/archive/105/159755; 1/31/2001
4. McClure, Scrambray, and Kurtz “Hacking Exposed” Network Security, URL: www.hackingexposed.com, 1999.
5. 2600 Magazine “The Hacker Quarterly” Computer Security and Hacking. 2001. URL: www.2600.com.