A Weakness in the 4.2BSD Unix† TCP/IP Software

Robert T. Morris

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The 4.2 Berkeley Software Distribution of the Unix operating system (4.2BSD for short) features an extensive body of software based on the "TCP/IP" family of protocols. In particular, each 4.2BSD system "trusts" some set of other systems, allowing users logged into trusted systems to execute commands via a TCP/IP network without supplying a password. These notes describe how the design of TCP/IP and the 4.2BSD implementation allow users on untrusted and possibly very distant hosts to masquerade as users on trusted hosts. Bell Labs has a growing TCP/IP network connecting machines with varying security needs; perhaps steps should be taken to reduce their vulnerability to each other.

February 25, 1985

† Unix is a Trademark of AT&T Bell Laboratories.

A Weakness in the 4.2BSD Unix† TCP/IP Software

Robert T. Morris

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

The Defense Department "TCP/IP" network protocol standard was designed in 1979 to implement an "internet": a group of networks, highly variable in reliability and speed, connected by computers acting as gateways. One of the more popular Unix TCP/IP implementations comes with the 4.2BSD system, used both within Bell Labs and on Defense Department networks. The 4.2BSD Unix TCP/IP software is very flexible and convenient, but places too much trust in a protocol which provides very little security. The attack described here requires no modifications to the system it runs on, and is not dependant on the hardware of the network involved.

TCP/IP conceptually divides into two layers, a "Transmission Control Protocol" and an "Internet Protocol". The IP layer[1] sends packets of data ("datagrams") from one host to another, via networks and gateways interconnecting them. TCP[2] supports a number of "ports" on each host running IP, providing reliable and flow controlled "virtual circuits" between these ports; TCP circuits are built on top of the IP datagram service. Each TCP or IP packet consists of a header full of control information followed by data; in the case of TCP the data is supplied by the user, while the data in an IP packet is a TCP packet. The important parts of the TCP header are a source port number, a destination port number, a sequence number, an acknowledgement number, and some flags. The port numbers identify which virtual circuit is involved, the sequence and acknowledgement numbers ensure that data is received in the correct order, and the flags affect the state of the virtual circuit. An IP header consists primarily of source and destination host identifiers; these are 32 bit numbers which uniquely indicate a host and a network. There is also a protocol number indicating which protocol layer (e.g. TCP) IP should direct the packet data to.

4.2BSD provides a remote execution "server", which listens for TCP connection requests on port 514. When such a request arrives at a machine, the server checks that the originating host is "trusted" by comparing the source host ID in the IP header to a list of trusted computers. If the source host is OK, the server reads a user id and a command to execute from the virtual circuit TCP provides. The weakness in this scheme is that the source host itself fills in the IP source host id, and there is no provision in 4.2BSD or TCP/IP to discover the true origin of a packet.

The ideal way to produce TCP/IP packets with incorrect source host id's would be to talk directly to the network involved. 4.2BSD provides no such network interface, so other means must be sought to forge packets from 4.2BSD systems. 4.2BSD does allow privileged users to send IP packets, though; with minimal effort the IP kernel code can be made to supply the correct protocol number (6), and an incorrect host id, in the IP header. The details involve creating a 4.2BSD "socket" with type "SOCK_RAW", and then writing on the kernel data structures to change the protocol number associated with "SOCK_RAW" to 6 (that of TCP) and to change the source host id. This requires privileges; however, it is likely that at least one system on a large network will be insecure enough to supply appropriate powers after a determined attack.

With appropriate access to IP, a user process can create and manage one end of a TCP circuit without using the TCP software in the Unix kernel. Each TCP header contains a checksum to detect inaccurate transmission. This checksum covers not only the TCP header and data, but also some of the IP header. Hence the user software must predict the contents of the IP header with which the kernel will encapsulate

† Unix is a Trademark of AT&T Bell Laboratories.
1. RFC 791, University of Southern California ISI, Marina del Ray, Cal. 90291
2. RFC 793, Sept 1981

the TCP packet. At this stage, a user process can send individual TCP packets.

The interesting TCP connection states are LISTEN, SYN_SENT, SYN_RCVD, and ESTABLISHED. Each TCP connection also maintains a sequence number as part of its state. The packet flags SYN, ACK, and RST (synchronize, acknowledge, and reset), as well as the packet acknowledgement number, affect the state. One end of a connection starts by sending a SYN and entering SYN_SENT; the other end starts out in LISTEN state. In the abbreviated state table following, each message is represented by a packet flag, the packet sequence number, the acknowledgement number, and possibly some data. Each state/event combination usually leads to a packet being sent, a state change, or possibly an error; each of the boxes in the diagram indicates a packet to be sent and a state to be entered. M means the sequence number of the packet just received; N means the sequence number remembered as part of the state of the TCP port. For instance, M would refer to the X in the received packet ACK,X,Y.

	SYN,X,Y	ACK,X,Y,data
LISTEN	SYN,N++,M+1 SYN_RCVD	error
SYN_SENT	ACK,N,M+1 ESTABLISHED	error
SYN_RCVD	RST,N,M error	ESTABLISHED
ESTABLISHED	RST,N,M error	ACK,N,M+data len ESTABLISHED (send data to user)

Data is sent by ACK,N,M,data when both sides of the connection are in the ESTABLISHED state, after which N is incremented by the length of the data. There are also other states and flags having to do with closing connections which are not relevant here.

4.2BSD maintains a global initial sequence number, which is incremented by 128 each second and by 64 after each connection is started; each new connection starts off with this number. When a SYN packet with a forged source is sent from a host, the destination host will send the reply to the presumed source host, not the forging host. The forging host must discover or guess what the sequence number in that lost packet was, in order to acknowledge it and put the destination TCP port in the ESTABLISHED state. Guessing the lost sequence number is easy when the destination runs 4.2BSD; one need only create a real connection, look in the kernel for the sequence number received, and add 64 to it. Once the forging program acknowledges this sequence number, the connection is fully set up and data may be sent, though not received, by the program.

Unfortunately, the SYN packet sent by the destination to the putative source does not just disappear. The supposed source sees it as a packet on a non-existent circuit, and sends a packet with a RST flag to the destination. This causes the destination to throw away the forged circuit. For instance: Host A sends a forged packet to B, claiming the source was C. B sends a SYN packet to C, and C sends a RST packet to B. B throws away the circuit that A is forging to it. The only ports on C that won't always generate RSTs in this situation are those which are waiting, or listening, for connections. Those listening ports have finite length queues of connections waiting to be set up; if this queue length is exceeded, the requesting SYN packet will be thrown away, but no reset will be generated. The originator is expected to resend the SYN packet after timing out. Note that original SYN packets and response SYN packets look the same. Thus it suffices for the forging process to claim that the packets are coming from a port on the supposed source that has a server listening for connections, and for the forger to flood that port with connection requests.

In summary, suppose the forging program is named A, its destination host is named B, the source to be forged is named C. The port on B involved is number 514, the remote execution server's port; A will forge packets from port 21 on host C, which is usually waiting for connections. The chain of events on A is as follows:

Swamp port 21 on C with connection requests.
Create a real connection to a port on B, and record the
    sequence number returned by B.
Create a raw IP socket, change its protocol to that of TCP,
    and change its source to C (by writing in the kernel).
Send a SYN packet from port 21 (supposedly on C) to port 514 on B.
    (A then sends a SYN to port 21 on C, which is silently ignored because
    C's queue for 21 is full.)
Send an ACK packet to B with the acknowledgement number equal to the
    sequence number previously recorded plus 64.
Send data to B, taking care to increment the sequence number each time
    by the amount of data sent. Port 514 expects a null, followed
    by a user name, followed by a command.
If all goes well, and B trusts C, B will execute the command.

Accuracy has been sacrificed for clarity, such as it is.

This scheme, with the details filled in, does in fact work fairly reliably. It allows machines on a TCP/IP network to run commands on any connected 4.2BSD system that "trusts" any other system. There are a number of possible defences. The sequence numbers that the forger must guess could be made very random; they are in a 32 bit word, so brute force search is unprofitable. However, the forger can ask for an arbitrarily large number of test connections to determine regularities in the random number algorithm; at best randomness will make the forger's job somewhat harder. A better approach might be to require that all networks IP uses supply genuine source host id's. This is network hardware dependent, and in any case will not work if gateways are involved. A workable solution might be to only trust hosts on the same physical network, and modify gateways to reject packets that claim to, but do not in fact, come from directly connected networks.