Simple Active Attack Against TCP

Laurent Joncheray

Merit Network, Inc.
4251 Plymouth Road, Suite C
Ann Arbor, MI 48105, USA
Phone: +1 (313) 936 2065
Fax: +1 (313) 747 3185
E-mail: lpj@merit.edu

Abstract

        This paper describes an active attack against the Transport
Control Protocol (TCP) which allows a cracker to redirect the TCP
stream through his machine thereby permitting him to bypass the protection
offered by such a system as a one-time password [skey] or
ticketing authentication [kerberos]. The TCP connection is
vulnerable to anyone with a TCP packet sniffer and generator located on
the path followed by the connection. Some schemes to detect this
attack are presented as well as some methods of prevention and some
interesting details of the TCP protocol behaviors.

1. Introduction

        Passive attacks using sniffers are becoming more and more
frequent on the Internet. The attacker obtains a user id and password
that allows him to logon as that user. In order to prevent such attacks
people have been using identification schemes such as one-time password
[skey] or ticketing identification [kerberos]. Though
they prevent password sniffing on an unsecure network these methods
are still vulnerable to an active attack as long as they neither
encrypt nor sign the data stream. [Kerberos also provides an
encrypted TCP stream option.] Still many people are complacent believing
that active attacks are very difficult and hence a lesser risk.

        The following paper describes an extremely simple active attack
which has been successfully used to break into Unix hosts and
which can be done with the same resources as for a passive sniffing
attack. [The attacks have been performed with a test software
and the users were aware of the attack.  Although we do not have any
knowledge of such an attack being used on the Internet, it may
be possible.] Some uncommon  behaviors of the TCP protocol are also
presented as well as some real examples and statistical studies of the
attack's impact on the network.  Finally some detection and prevention
schemes are explained. In order to help any reader unfamiliar with the
subtleties of the TCP protocol the article starts with a short
description of TCP.

        The reader can also refers to another attack by R. Morris
presented in [morris85]. Though the following attack is related
to Morris' one, it is more widely usable on any TCP connection.
In section 7 we present and compare this attack with
the present one.

        The presentation of the attack will be divided into three parts:
the ``Established State'' which is the state where the session is open
and data is exchanged; the set up (or opening) of such a session; and
finally some real examples.

2. Established State

2.1 The TCP protocol

This section offers a short description of the TCP protocol.
For more details the reader can refer to [rfc793]. TCP provides a
full duplex reliable stream connection between two end points. A
connection is uniquely defined by the quadruple (IP address of sender,
TCP port number of the sender, IP address of the receiver, TCP port
number of the receiver). Every byte that is sent by a host is marked with a
sequence number (32 bits integer) and is acknowledged by the receiver
using this sequence number. The sequence number for the first byte sent
is computed during the connection opening. It changes for any new
connection based on rules designed to avoid reuse of the same sequence
number for two different sessions of a TCP connection.

We shall assume in this document that one point of the
connection acts as a server (for instance a telnet server) and the
other as the client.  The following terms will be used:

SVR_SEQ: sequence number of the next byte to be sent
by the server;
SVR_ACK: next byte to be received by the server
(the sequence number of the last byte received plus one);
SVR_WIND: server's receive window;
CLT_SEQ: sequence number of the next byte to be sent by
the client;
CLT_ACK: next byte to be received by the client;
CLT_WIND: client's receive window;

At the beginning when no data has been exchanged we have
SVR_SEQ = CLT_ACK and CLT_SEQ = SVR_ACK. These equations
are also true when the connection is in a 'quiet' state (no data being
sent on each side).  They are not true during transitory states when
data is sent.  The more general equations are:

CLT_ACK <= SVR_SEQ <= CLT_ACK + CLT_WIND
SVR_ACK <= CLT_SEQ <= SVR_ACK + SVR_WIND

The TCP packet header fields are:

Source Port:            The source port number;
Destination Port:       The destination port number;
Sequence number:        The sequence number of the first
                        byte in this packet;
Acknowledgment Number:  The expected sequence number
                             of the next byte to be received;
Data Offset:            Offset of the data in the packet;
Control Bits:

        URG:  Urgent Pointer;
        ACK:  Acknowledgment;
        PSH:  Push Function;
        RST:  Reset the connection;
        SYN:  Synchronize sequence numbers;
        FIN:  No more data from sender;

Window:         Window size of the sender;
Checksum:       TCP checksum of the header and data;
Urgent Pointer: TCP urgent pointer;
Options:              TCP options;

 - SEG_SEQ will refer to the packet sequence number (as
seen in the header).
 - SEG_ACK will refer to the packet acknowledgment number.
 - SEG_FLAG will refer to the control bits.

On a typical packet sent by the client (no retransmission) SEG_SEQ is set

to CLT_SEQ, SEG_ACK to CLT_ACK.

        TCP uses a ``three-way handshake'' to establish a new
connection. If we suppose that the client initiates the connection to
the server and that no data is exchanged, the normal packet exchange
is (C.f. Figure 1):

 - The connection on the client side is on the CLOSED state.
The one on the server side is on the LISTEN state.
 - The client first sends its initial sequence number and sets the SYN bit:

        SEG_SEQ  =  CLT_SEQ_0,
        SEG_FLAG =  SYN

Its state is now SYN-SENT
 - On receipt of this packet the server acknowledges the client sequence
number, sends its own initial sequence number and sets the SYN bit:

        SEG_SEQ  =  SVR_SEQ_0,
        SEQ_ACK  =  CLT_SEQ_0+1,
        SEG_FLAG =  SYN

and set

        SVR_ACK=CLT_SEQ_0+1

Its state is now SYN-RECEIVED
 - On receipt of this packet the client acknowledges the server
sequence number:

        SEG_SEQ  =  CLT_SEQ_0+1,
        SEQ_ACK  =  SVR_SEQ_0+1

and sets CLT_ACK=SVR_SEQ_0+1
Its state is now ESTABLISHED
 - On receipt of this packet the server enters the ESTABLISHED state. We now
have:

        CLT_SEQ  =  CLT_SEQ_0+1
        CLT_ACK  =  SVR_SEQ_0+1
        SVR_SEQ  =  SVR_SEQ_0+1
        SVR_ACK  =  CLT_SEQ_0+1

```
Server                                          Client
LISTEN                                          CLOSED

                <-  SYN,
                    CLT_SEQ_0

LISTEN                                          SYN-SENT

                SYN,        ->
                SVR_SEQ_0,
                CLT_SEQ_0+1

SYN-RECEIVED                                    ESTABLISHED

                                                SVR_SEQ = CLT_SEQ_0 + 1
                                                CLT_ACK = SVR_SEQ_0 + 1

                <-  ACK,
                    CLT_SEQ_0 + 1
                    SVR_SEQ_0+1
```

ESTABLISHED

SVR_SEQ = SVR_SEQ_0 + 1
SVR_ACK = CLT_SEQ_0 + 1


                    figure 1: Example of a connection opening

        Closing a connection can be done by using the FIN or the RST
flag.  If the RST flag of a packet is set the receiving host enters the
CLOSED state and frees any resource associated with this instance of
the connection.  The packet is not acknowledged. Any new incoming
packet for that connection will be dropped.

If the FIN flag of a packet is set the receiving host enters the
CLOSE-WAIT state and starts the process of gracefully closing the
connection.  The detail of that procces is beyond the scope of this
document. The reader can refer to [rfc793] for further details.

        In the preceding example we specifically avoided any unusual cases
such as out-of-band packets, retransmission, loss of packet, concurrent
opening, etc...  These can be ignored in this simple study of the
attack.

        When in ESTABLISHED state, a packet is acceptable if its
sequence number falls within the expected segment

        [SVR_ACK, SVR_ACK + SVR_WIND]

(for the server) or

        [CLT_ACK, CLT_ACK + CLT_WIND]

(for the client). If the sequence number is beyond those limits the
packet is dropped and a acknowledged packet will be sent using the
expected sequence number.  For example if

        SEG_SEQ  =  200,
        SVR_ACK  =  100,
        SVR_WIND  =  50

Then SEG_SEQ > SVR_ACK + SVR_WIND. The server
forms a ACK packet with

        SEG_SEQ  =  SVR_SEQ
        SEG_ACK  =  SVR_ACK

which is what the server expects to see in the packet.

2.2 A desynchronized state

        The term ``desynchronized state'' will refer to the connection
when both sides are in the ESTABLISHED state, no data is being sent
(stable state), and

        SVR_SEQ  !=  CLT_ACK
        CLT_SEQ  !=  SVR_ACK


        This state is stable as long as no data is sent. If some data
is sent two cases can occur:

 - If CLT_SEQ < SVR_ACK + SVR_WIND and
CLT_SEQ > SVR_ACK the packet is acceptable, the data may be stored

for later use (depending on the implementation) but not sent to the
user since the beginning of the stream (sequence number SVR_ACK) is
missing.
 - If CLT_SEQ > SVR_ACK + SVR_WIND or CLT_SEQ <
SVR_ACK the packet is not acceptable and will be dropped. The data is
lost.

        In both case data exchange is not possible even if the state
exists.

2.3 The attack

        The proposed attack consists of creating a desynchronized state
on both ends of the TCP connection so that the two points cannot exchange data
any longer. A third party host is then used to create acceptable packets
for both ends which mimics the real packets.

        Assume that the TCP session is in a desynchronized state and that
the client sends a packet with

        SEG_SEQ  =  CLT_SEQ
        SEG_ACK  =  CLT_ACK

Since CLT_SEQ != SVR_ACK the data will not be accepted and the
packet is dropped.  The third party then sends the same packet but
changes the SEG_SEQ and SEG_ACK (and the checksum) such that

        SEG_SEQ  =  SVR_ACK,
        SEG_ACK  =  SVR_SEQ

which is acceptable by the server. The data is processed by the server.

If CLT_TO_SVR_OFFSET refers to SVR_ACK - CLT_SEQ and
SVR_TO_CLT_OFFSET refers to CLT_ACK - SVR_SEQ then the first party
attacker has to rewrite the TCP packet from the client to the server as:

        SEG_SEQ <- SEG_SEQ + CLT_TO_SVR_OFFSET
        SEG_ACK <- SEG_ACK - SVR_TO_CLT_OFFSET

Considering that the attacker can listen to any packet exchanged between
the two points and can forge any kind of IP packet (therefore masquerading as
either the client or the server) then everything acts as if the connection
goes through the attacker machine. This one can add or remove any data to
the stream. For instance if the connection is a remote login using telnet
the attacker can include any command on behalf of the user
("echo merit.edu lpj > ~/.rhosts" is an example of such a command)
and filter out any unwanted echo so that the user will not
be aware of the intruder.
Of course in this case CLT_TO_SVR_OFFSET and SVR_TO_CLT_OFFSET
have to change. The new values are let as an exercise for the
reader. [One can turn off
the echo in the telnet connection in order to avoid the burden of filtering
the output. The test we did showed up a bug in the current telnet
implementation (or maybe in the telnet protocol itself). If a TCP packet
contains both
IAC DONT ECHO and IAC DO ECHO the telnet processor will answer with
IAC WONT ECHO and IAC WILL ECHO. The other end point will acknowledge
IAC DONT ECHO and IAC DO ECHO etc... creating an endless loop.]

2.4 ``TCP Ack storm''

        A flaw of the attack is the generation of a lot of TCP ACK
packets. When receiving an unacceptable packet the host acknowledges it
by sending the expected sequence number (As the Acknolegement number.

C.f. introduction about TCP)
and using its own sequence number. This packet is itself unacceptable and
will generate an acknowledgement packet which in turn will generate
an acknowledgement packet etc... creating a supposedly endless loop for
every data packet sent.

Since these packets do not carry data they are not retransmitted
if the packet is lost. This means that if one of the packets
in the loop is dropped then the loop ends. Fortunately (or
unfortunately?) TCP uses IP on an unreliable network layer with a non
null packet loss rate, making an end to the loops. Moreover the more
packets the network drops, the shorter is the Ack storm (the loop). We
also notice that these loops are self regulating: the more loops we
create the more traffic we get, the more congestion and packet drops we
experience and the more loops are killed.

The loop is created each time the client or the server sends data.
If no data is sent no loop appears. If data is sent and no attacker is there
to acknowledge the data then the data will be retransmitted, a storm
will be created for each retransmission, and eventually the connection
will be dropped since no ACK of the data is sent. If the attacker acknowledges
the data then only one storm is produced (in practice the attacker often
missed the data packet due to the load on the network, and acknowledge the
first of subsequent retransmission).

The attack uses the second type of packet described in
Section 2.2. The first case in which the data is stored by
the receiver for later processing has not been tested. It has the
advantage of not generating the ACK storm but on the other hand it may be
dangerous if the data is actually processed. It is also difficult to
use with small window connections.

3. Setup of the session

This paper presents two methods for desynchronizing a TCP connection.
Others can be imagined but will not be described here. We suppose that the
attacker can listen to every packet sent between the two end points.

3.1 Early desynchronization

This method consists of breaking the connection in its early setup stage
on the server side and creating a new one with different sequence number.
Here is the process (Figure 2 summarizes this process)

- The attacker listens for a SYN/ACK packet from the server to
the client (stage 2 in the connection set up).

- On detection of that packet the attacker sends the server
a RST packet and then a SYN packet with exactly the same parameters
(TCP port) but a different sequence number (referred to as ATK_ACK_0 in
the rest of the paper).

- The server will close the first connection when it
receives the RST packet and then reopens a new one on the same port but
with a different sequence number (SVR_SEQ_0') on receipt of the SYN
packet. It sends back a SYN/ACK packet to the client.

- On detection of that packet the attacker sends the server a
ACK packet. The server switches to the ESTABLISHED state.

- The client has already switched to the ESTABLISHED state when it
receives the first SYN/ACK packet from the server.

```
Server                                        Client

LISTEN                                        CLOSED

                    <-  SYN,
                        CLT_SEQ_0

SYN-RECEIVED                                  SYN-SENT

                    SYN,          ->
                    SVR_SEQ_0,
                    CLT_SEQ_0+1


                                              ESTABLISHED

                                              SVR_SEQ = CLT_SEQ_0 + 1
                                              CLT_ACK = SVR_SEQ_0 + 1

                    <=  RST,
                        CLT_SEQ_0 + 1

CLOSED

                    <=  SYN,
                        ATK_SEQ_0


                    SYN,           ->
                    SVR_SEQ_0',
                    ATK_SEQ_0 + 1

SYN-RECEIVED

                    <=  SYN,
                        ATK_SEQ_0 + 1,
                        SVR_SEQ_0' + 1


ESTABLISHED

SVR_SEQ = SVR_SEQ_0' + 1
SVR_ACK = ATK_SEQ_0 + 1
```
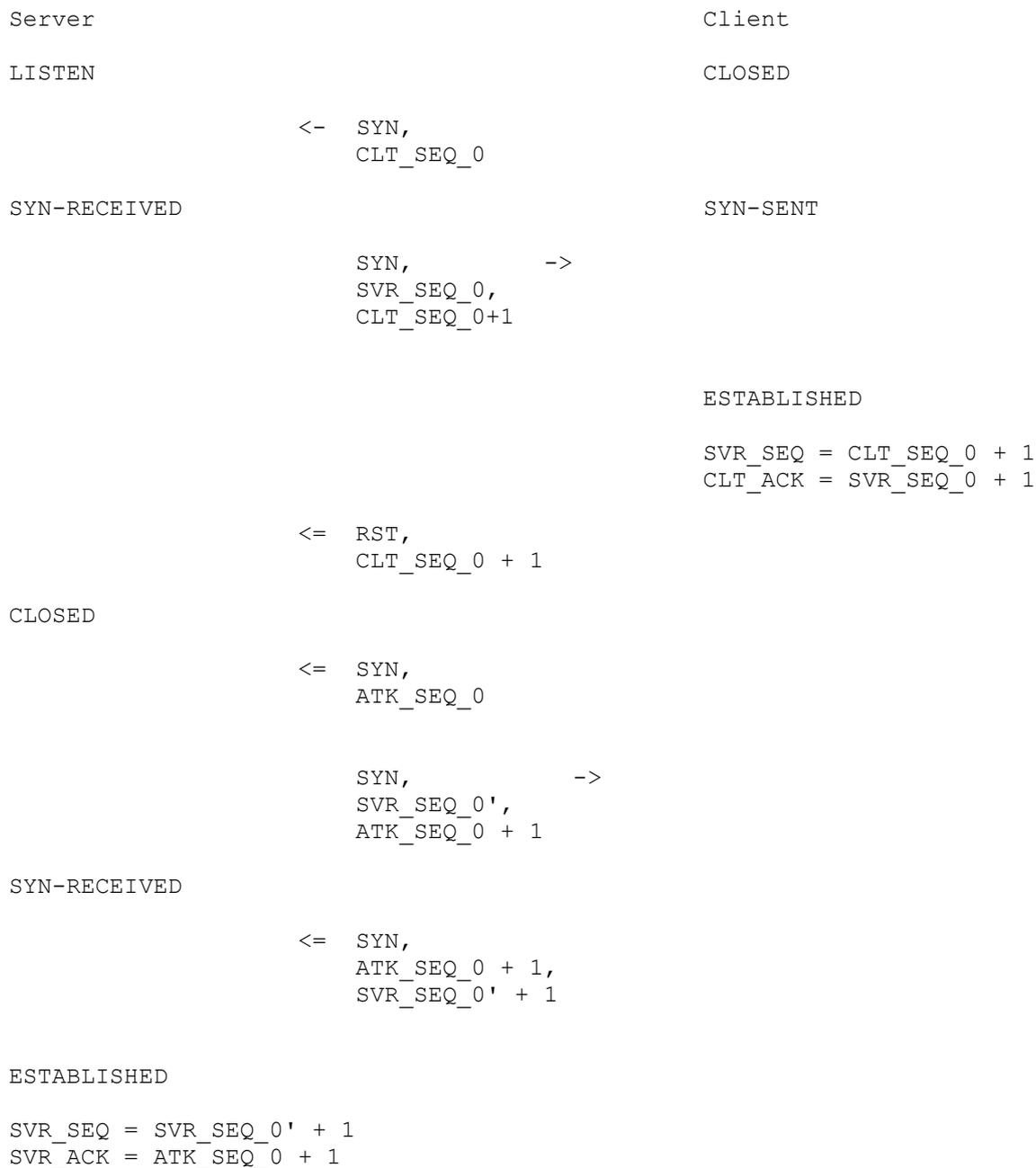
        Figure 2: A attack scheme. The attacker's packets are marked with <=

        This diagram does not show the unacceptable acknowledgement packet
exchanges. Both ends are in the desynchronized ESTABLISHED state now.

        SVR_TO_CLT_OFFSET = SVR_SEQ_0 - SVR_SEQ_0'

is fixed by the server.

        CLT_TO_SVR_OFFSET = ATK_SEQ_0 - CLT_SEQ_0

is fixed by the attacker.

        The success of the attack relies on the correct value being chosen
for CLT_TO_SVR_OFFSET. Wrong value may make the
client's packet acceptable and can produce unwanted effects.

3.2 Null data desynchronization

This method consists for the attacker in sending a large amount
of data to the server and to the client. The data sent shouldn't affect
nor be visible to the client or sever, but will put both end of the TCP
session in the desynchronized state.

        The following scheme can be used with a telnet session:

         - The attacker watchs the session without interfering.
         - When appropriate the attacker sends a large amount of ``null
data'' to the server. ``Null data'' refers to data that will not affect
anything on the server side besides changing the TCP acknowledgment number.
[For instance with a telnet session the attacker sends ATK_SVR_OFFSET
bytes consisting of the sequence IAC NOP IAC NOP... Every two
bytes IAC NOP will be interpreted by the telnet daemon, removed from
the stream of data and nothing will be affected. [The telnet
protocol [telnet] defines the NOP command as ``No Operation''. In
other words, do nothing, just ignore those bytes.] Now the Server has

        SVR_ACK = CLT_SEQ + ATK_SVR_OFFSET

which of course is desynchronized.
         - The attacker does the same thing with the client.


        The method is useful if the session can carry ``null data''. The
time when the attacker sends that data is also very difficult to determine
and may cause some unpredictable side effects.

4. Examples

        The following logs are provided by running a hacked version of
tcpdump [tcpdump] on the local ethernet where the client resides.
Comments are preceded by `##'.

        The first example is a normal telnet session opening between 35.42.1.56
(the client) and 198.108.3.13 (the server).

        ## The client sends a SYN packet, 1496960000 is its initial sequence nu
mber.
11:07:14.934093 35.42.1.56.1374 > 198.108.3.13.23: S 1496960000:1496960000(0) w
in 4096
        ## The server answers with its initial sequence number and the SYN flag
.
11:07:14.936345 198.108.3.13.23 > 35.42.1.56.1374: S 1402880000:1402880000(0) a
ck 1496960001 win 4096
        ## The client acknowledges the SYN packet.
11:07:14.937068 35.42.1.56.1374 > 198.108.3.13.23: . 1496960001:1496960001(0) a
ck 1402880001 win 4096
        ## Now the two end points are in the ESTABLISHED state.
        ## The client sends 6 bytes of data.
11:07:15.021817 35.42.1.56.1374 > 198.108.3.13.23: P 1496960001:1496960007(6)
        ack 1402880001 win 4096 255 253 /C 255 251 /X
[...
        ## The rest of the log is the graceful closing of the connection
11:07:18.111596 198.108.3.13.23 > 35.42.1.56.1374: F 1402880059:1402880059(0) a
ck 1496960025 win 4096
11:07:18.112304 35.42.1.56.1374 > 198.108.3.13.23: . 1496960025:1496960025(0) a
ck 1402880060 win 4096
11:07:18.130610 35.42.1.56.1374 > 198.108.3.13.23: F 1496960025:1496960025(0) a
ck 1402880060 win 4096
11:07:18.132935 198.108.3.13.23 > 35.42.1.56.1374: . 1402880060:1402880060(0) a
ck 1496960026 win 4095

        The next example is the same session with an intrusion by the

attacker. The desynchronized state is created in the early stage of
the session (subsection 3.1). The attacker will add the
command 'ls;' to the stream of data.  The user uses skey to
identify himself to the server. From the user's point of view the
session looks like this:

```
<lpj@homefries: 1> telnet 198.108.3.13
Trying 198.108.3.13 ...
Connected to 198.108.3.13.
Escape character is '^'.

SunOS UNIX (_host)

login: lpj
s/key 70 cn33287
(s/key required)
Password:
Last login: Wed Nov 30 11:28:21 from homefries.merit.edu
SunOS Release 4.1.3_U1 (GENERIC) #2: Thu Jan 20 15:58:03 PST 1994
(lpj@_host: 1) pwd
Mail/           mbox            src/
elm*            resize*         traceroute*
/usr/users/lpj
(lpj@_host: 2) history
     1  13:18   ls ; pwd
     2  13:18   history
(lpj@_host: 3) logoutConnection closed by foreign host.
<lpj@homefries: 2>
```

The user types only one command 'pwd' and then asks for the history of
the session. The history shows that a ls' has also being issued.
The ls command produces an output which has not been filtered.
The following log shows the TCP packet exchanges between the client and
the server. Unfortunately some packets are missing from this log because
they have been dropped by the sniffer's ethernet interface driver. One
must see that log like a snapshot of a few instants of the exchange
more than the full transaction log. The attacker's window size has been
set to uncommon values (400, 500, 1000) in order to make its packets
more easily traceable. The attacker is on 35.42.1, three hops away from the
server, on the path from the client to the server. The names and addresses
of the hosts have been changed for security reasons.

```
        ## The client sends a SYN packet, 896896000 is its initial sequence num
ber.
11:25:38.946119 35.42.1.146.1098 > 198.108.3.13.23: S 896896000:896896000(0) wi
n 4096
        ## The server answers with its initial sequence number (1544576000) and
 the SYN flag.
11:25:38.948408 198.108.3.13.23 > 35.42.1.146.1098: S 1544576000:1544576000(0)
ack 896896001 win 4096
        ## The client acknowledges the SYN packet. It is in the ESTABLISHED sta
te now.
11:25:38.948705 35.42.1.146.1098 > 198.108.3.13.23: . 896896001:896896001(0) ac
k 1544576001 win 4096
        ## The client sends some data
11:25:38.962069 35.42.1.146.1098 > 198.108.3.13.23: P 896896001:896896007(6)
        ack 1544576001 win 4096 255 253 /C 255 251 /X
        ## The attacker resets the connection on the server side
11:25:39.015717 35.42.1.146.1098 > 198.108.3.13.23: R 896896101:896896101(0) wi
n 0
        ## The attacker reopens the connection with an initial sequence number
of 601928704
11:25:39.019402 35.42.1.146.1098 > 198.108.3.13.23: S 601928704:601928704(0) wi
n 500
```

```
        ## The server answers with a new initial sequence number (1544640000) a
nd the SYN flag.
11:25:39.022078 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0)
ack 601928705 win 4096
        ## Since the last packet is unacceptable for the client, it acknowledge
s it
        ## with the expected sequence number (1544576001)
11:25:39.022313 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
        ## Retransmission to the SYN packet triggered by the unacceptable last
packet
11:25:39.023780 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0)
ack 601928705 win 4096
        ## The ACK storm loop
11:25:39.024009 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
11:25:39.025713 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0)
ack 601928705 win 4096
11:25:39.026022 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
[...
11:25:39.118789 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0)
ack 601928705 win 4096
11:25:39.119102 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
11:25:39.120812 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0)
ack 601928705 win 4096
11:25:39.121056 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
        ## Eventually the attacker acknowledges the server SYN packet with the
attacker's new
        ## sequence number (601928705). The data in this packet is the one prev
iously
        ## sent by the client but never received.
11:25:39.122371 35.42.1.146.1098 > 198.108.3.13.23: . 601928705:601928711(6)
        ack 1544640001 win 400 255 253 /C 255 251 /X
        ## Some ACK storm
11:25:39.124254 198.108.3.13.23 > 35.42.1.146.1098: . 1544640001:1544640001(0)
ack 601928711 win 4090
11:25:39.124631 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
11:25:39.126217 198.108.3.13.23 > 35.42.1.146.1098: . 1544640001:1544640001(0)
ack 601928711 win 4090
11:25:39.126632 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ac
k 1544576001 win 4096
[...
11:25:41.261885 35.42.1.146.1098 > 198.108.3.13.23: . 601928728:601928728(0) ac
k 1544640056 win 1000
        ## A retransmission by the client
11:25:41.422727 35.42.1.146.1098 > 198.108.3.13.23: P 896896018:896896024(6)
        ack 1544576056 win 4096 255 253 /A 255 252 /A
11:25:41.424108 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0)
ack 601928728 win 4096
[...
11:25:42.323262 35.42.1.146.1098 > 198.108.3.13.23: . 896896025:896896025(0) ac
k 1544576059 win 4096
11:25:42.324609 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0)
ack 601928728 win 4096
        ## The user ID second character.
11:25:42.325019 35.42.1.146.1098 > 198.108.3.13.23: P 896896025:896896026(1)
        ack 1544576059 win 4096 p
11:25:42.326313 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0)
ack 601928728 win 4096
[...
```

```
11:25:43.241191 35.42.1.146.1098 > 198.108.3.13.23: . 601928731:601928731(0) ac
k 1544640060 win 1000
        ## Retransmission
11:25:43.261287 198.108.3.13.23 > 35.42.1.146.1098: P 1544640059:1544640061(2)
        ack 601928730 win 4096 l p
11:25:43.261598 35.42.1.146.1098 > 198.108.3.13.23: . 896896027:896896027(0) ac
k 1544576061 win 4096
[...
11:25:43.294192 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0)
ack 601928730 win 4096
11:25:43.922438 35.42.1.146.1098 > 198.108.3.13.23: P 896896026:896896029(3)
        ack 1544576061 win 4096 j /M /@
11:25:43.923964 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0)
ack 601928730 win 4096
[...
11:25:43.957528 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0)
ack 601928730 win 4096
        ## The attacker rewrites the packet sent by the server containing the s
key challenge
11:25:44.495629 198.108.3.13.23 > 35.42.1.146.1098: P 1544576064:1544576082(18)

        ack 896896029 win 1000 s / k e y   7 0   c n 3 3 2 8 7 /M /J
11:25:44.502533 198.108.3.13.23 > 35.42.1.146.1098: P 1544576082:1544576109(27)

        ack 896896029 win 1000 ( s / k e y   r e q u i r e d ) /M /J P a s s w
o r d :
11:25:44.522500 35.42.1.146.1098 > 198.108.3.13.23: . 896896029:896896029(0) ac
k 1544576109 win 4096
[...
11:25:44.558320 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0)
ack 601928733 win 4096
        ## Beginning of the skey password sent by the user (client)
11:25:57.356323 35.42.1.146.1098 > 198.108.3.13.23: P 896896029:896896030(1)
        ack 1544576109 win 4096 T
11:25:57.358220 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0)
ack 601928733 win 4096
[...
11:25:57.412103 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0)
ack 601928733 win 4096
        ## Echo of the beginning of the skey password sent by the server
11:25:57.412456 35.42.1.146.1098 > 198.108.3.13.23: P 601928733:601928734(1)
        ack 1544640109 win 1000 T
11:25:57.412681 35.42.1.146.1098 > 198.108.3.13.23: . 896896030:896896030(0) ac
k 1544576109 win 4096
[...
11:25:57.800953 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0)
ack 601928734 win 4096
        ## The attacker rewrites the skey password packet
11:25:57.801254 35.42.1.146.1098 > 198.108.3.13.23: P 601928734:601928762(28)
        ack 1544640109 win 1000 A U T   S H I M   L O F T   V A S E   M O O   R
 I D /M /@
11:25:57.801486 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ac
k 1544576109 win 4096
[...
11:25:58.358275 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ac
k 1544576109 win 4096
11:25:58.360109 198.108.3.13.23 > 35.42.1.146.1098: P 1544640263:1544640278(15)

        ack 601928762 win 4096 ( l p j @ _ r a d b :   1 )
11:25:58.360418 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ac
k 1544576109 win 4096
[...
11:26:00.919976 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ac
k 1544576278 win 4096
```

```
        ## The 'p' of the 'pwd' command typed by the user.
11:26:01.637187 35.42.1.146.1098 > 198.108.3.13.23: P 896896058:896896059(1)
        ack 1544576278 win 4096 p
11:26:01.638832 198.108.3.13.23 > 35.42.1.146.1098: . 1544640278:1544640278(0)
ack 601928762 win 4096
[...
11:26:03.183200 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ac
k 1544576280 win 4096
11:26:03.921272 35.42.1.146.1098 > 198.108.3.13.23: P 896896060:896896063(3)
        ack 1544576280 win 4096 d /M /@
11:26:03.922886 198.108.3.13.23 > 35.42.1.146.1098: . 1544640283:1544640283(0)
ack 601928767 win 4096
[...
11:26:04.339186 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ac
k 1544576280 win 4096
11:26:04.340635 198.108.3.13.23 > 35.42.1.146.1098: P 1544640288:1544640307(19)

        ack 601928770 win 4096 M a i l / /I /I m b o x /I /I s r c / /M /J
11:26:04.342872 198.108.3.13.23 > 35.42.1.146.1098: P 1544640307:1544640335(28)

        ack 601928770 win 4096 e l m * /I /I r e s i z e * /I /I t r a c e r o
u t e * /M
 /J
11:26:04.345480 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ac
k 1544576280 win 4096
11:26:04.346791 198.108.3.13.23 > 35.42.1.146.1098: P 1544640335:1544640351(16)

        ack 601928770 win 4096 / u s r / u s e r s / l p j /M /J
11:26:04.347094 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ac
k 1544576280 win 4096
11:26:04.348402 198.108.3.13.23 > 35.42.1.146.1098: P 1544640351:1544640366(15)

        ack 601928770 win 4096 ( l p j @ _ r a d b :   2 )
11:26:04.378571 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ac
k 1544576280 win 4096
[...
11:26:09.791045 35.42.1.146.1098 > 198.108.3.13.23: P 601928773:601928775(2)
        ack 1544640369 win 1000 t o
11:26:09.794653 198.108.3.13.23 > 35.42.1.146.1098: P 1544640369:1544640371(2)
        ack 601928775 win 4096 t o
11:26:09.794885 35.42.1.146.1098 > 198.108.3.13.23: . 896896068:896896068(0) ac
k 1544576366 win 4096
[...
11:26:12.420397 35.42.1.146.1098 > 198.108.3.13.23: P 896896068:896896072(4)
        ack 1544576368 win 4096 r y /M /@
11:26:12.422242 198.108.3.13.23 > 35.42.1.146.1098: . 1544640371:1544640371(0)
ack 601928775 win 4096
[...
11:26:12.440765 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ac
k 1544576368 win 4096
        ## The 'ry' of the 'history' command sent by the client
11:26:16.420287 35.42.1.146.1098 > 198.108.3.13.23: P 896896068:896896072(4)
        ack 1544576368 win 4096 r y /M /@
11:26:16.421801 198.108.3.13.23 > 35.42.1.146.1098: . 1544640371:1544640371(0)
ack 601928775 win 4096
[...
11:26:16.483943 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ac
k 1544576368 win 4096
        ## The same packet rewritten by the attacker.
11:26:16.505773 35.42.1.146.1098 > 198.108.3.13.23: P 601928775:601928779(4)
        ack 1544640371 win 1000 r y /M /@
        ## answer to the history command sent by the server. We can notice the
'ls ;' inclusion
        ## before the 'pwd'
```

```
11:26:16.514225 198.108.3.13.23 > 35.42.1.146.1098: P 1544640371:1544640437(66)

        ack 601928779 win 4096 r y /M /@ /M /J        1 /I 1 1 : 2 8 /I l s
   ;    p w
        d /M /J          2 /I 1 1 : 2 8 /I /@ /@ /@ L /@ /@ /@ T . 220 167 168
 /@ /G
      /@ /@ /@ /X /@ /H 137 148 /@ /@
11:26:16.514465 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ac
k 1544576368 win 4096
[...
11:26:16.575344 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ac
k 1544576368 win 4096
        ## The same packet rewritten by the attacker.
11:26:16.577183 198.108.3.13.23 > 35.42.1.146.1098: P 1544576368:1544576434(66)

        ack 896896072 win 1000 r y /M /@ /M /J        1 /I 1 1 : 2 8 /I l s
   ;    p w
d /M /J            2 /I 1 1 : 2 8 /I /@ /@ /@ L /@ /@ /@ T . 220 167 168 /@ /H /
@ /@ /@
      /X /@ /H 137 148 /@ /@
11:26:16.577490 198.108.3.13.23 > 35.42.1.146.1098: . 1544640437:1544640437(0)
ack 601928779 win 4096
[...
        ## The user log out.
11:26:20.236907 35.42.1.146.1098 > 198.108.3.13.23: P 601928781:601928782(1) ac
k 1544640437 win 1000 g
11:26:20.247288 198.108.3.13.23 > 35.42.1.146.1098: . 1544576438:1544576438(0)
ack 896896074 win 1000
11:26:20.253500 198.108.3.13.23 > 35.42.1.146.1098: P 1544576435:1544576436(1)
ack 896896074 win 1000 o
11:26:20.287513 198.108.3.13.23 > 35.42.1.146.1098: P 1544640439:1544640440(1)
ack 601928782 win 4096 g
11:26:20.287942 35.42.1.146.1098 > 198.108.3.13.23: P 896896075:896896076(1) ac
k 1544576436 win 4096 o
11:26:20.289312 198.108.3.13.23 > 35.42.1.146.1098: . 1544640440:1544640440(0)
ack 601928782 win 4096
11:26:20.289620 35.42.1.146.1098 > 198.108.3.13.23: . 896896076:896896076(0) ac
k 1544576436 win 4096
```

        Almost all of the packets with the ACK flag set but with no
data are acknowledgement of unacceptable packets. A lot of
retransmission occurs due to the load on the network and on the
attacker host created by the ACK storm. The real log (including all ACK
packets) is about 3000 lines long whereas the one shown here has been
stripped to about 100 lines. A lot of packets have also been lost and do
not show up in this log. The data collected during the test shows that
one real packet sent can generate between 10 and 300 empty Ack
packets.  Those numbers are of course highly variable.

5. Detection and Side Effects

        Several flaws of that attack can be used to detect it. Three will
be described here but one can imagine some other ways to detect the intrusion.


 - Desynchronized state detection.  By comparing the sequence
numbers of both ends of the connection the user can tell if the
connection is in the desynchronized state. This method is feasible if
we assume that the sequence numbers can be transmitted through the TCP
stream without being compromised (changed) by the attacker.

|           | Local Ethernet |        | Transit Ethernet | |
|-----------|----------------|--------|------------------|------|
| Total TCP/s | 80-100 | (60-80) | 1400 | (87) |
| Total Ack | 25-75 | (25-45) | 500 | (35) |

| | | | | |
|---|---|---|---|---|
| Total Telnet | 10-20 | (10-25) | 140 | (10) |
| Total Telnet Ack | 5-10 | (45-55) | 45 | (33) |

Table 1: Percentage of ACK packets without the attack.

 - Ack storm detection.
        Some statistics on the TCP traffic conducted on our local
ethernet segment outside the attack show that the average ratio of ACK
without data packets per total telnet packets is around 45%. On a more
loaded transit ethernet the average is about 33% (C.f Table 1)

        The total number of TCP packets as well as the total number of
ACK and telnet packets fluctuate a lot on the local ethernet. The table shows
the limits. The percentage of ACK telnet packets is very stable, around 45%.
This can be explained by the fact that the telnet session is an interactive
session and every character typed by the user must be echoed and acknowledged.
The volume of exchanged data is very small each packet usually contains one
character or one text line.

        The data for the transit ethernet is very consistent. Due to the
high load on that segment a few packets may have been dropped by the
collecting host.

        When the attack is conducted some of these figures change. The
next table shows the results for two types of session. The data has been
collected on the local ethernet only.

        In Table 2 the `Local connection' is a
session with a host at a few IP hops from the client. The Round Trip
Delay (RTD) is approximately 3ms and the actual number of hops is 4.
The 'Remote connection' is a session with a RTD of about 40ms and 9
hops away. In the first case the attack is clearly visible. Even if
it's very fluctuant, the percentage of TCP ACK is near 100%. Almost
all of the traffic is acknowledgement packets.

        In the second case the detection of the attack is less obvious.
The data has to be compared with the first column of
Table 1 (local traffic).  The percentage of TCP ACK
slightly increases but not significantly.  One can explain this result
by the long RTD which decreases the rate of ACK packets sent. The
underlying network is also used to experience between a 5% and 10%
packet loss which helps in breaking the ACK loop.

| | Local connection | | Remote connection | |
|---|---|---|---|---|
| Total Telnet | 80-400 | (60-85) | 30-40 | (30-35) |
| Total Telnet Ack | 75-400 | (90-99) | 20-25 | (60-65) |

        Percentage of ACK packets during an attack.

         - Increase of the packet loss and retransmission for that
particular session. Though no data is available to enlighten us on that
behavior the log produced during the attack shows an unusually high
level of packet loss and so retransmission. Therefore this implies a
deterioration of the response time for the user. The packet loss
increase is caused by:

          - The extra load of the network due to the ACK storms.
          - The packet dropped by the sniffer of the attacker. The drops tend
          to increase as the load on the network increases.

 - Some unexpected connection reset.
The following behavior has not been fully investigated since the
attacker program developed was to try the validity of the concept more
than making the attack transparent to the client and server. These are

likely to disappear with a more sophisticated attacker program. The user
can experience a connection reset of its session at the early stage of
the connection if the protocol of the attack is not correctly executed.
A loss of the attacker's RST or SYN packets may leave the server side of the
connection in a undefined state (usually CLOSED or SYN-RECEIVED) and
may make the client packets acceptable. About 10% of the attacks performed
were unsuccessful, ending either by a connection close (very visible)
or a non-desynchronized connection (the attacker failed to redirect
the stream).

Some side effects and notes about TCP and the attack.

        - TCP implementation.
        The desynchronization process described here failed on certain
TCP implementations. According to [rfc793] a RST packet is not
acknowledged and just destroys the TCB. Some TCP implementations do
when in a certain state acknowledge the RST packet by sending back a
RST packet. When the attacker sends the RST packet to the server the
RST is sent back to the client which closes its connection and ends the
session. Other desynchronization mechanisms may be investigated which
do not reset the connection.
        - The client and the attacker were always on the same
ethernet segment when performing the test. This makes the attack more
difficult to run because of a high load on that segment. The collision
rate increases and the attacker's sniffer buffer are overflowed by the
traffic.
        - One can think of just watching the session and
sending some data to the server, without caring about creating the
desynchronized state and forwarding the TCP packets. Though it will
succeed in corrupting the host that approach is likely to be detected early
by the user. Indeed the TCP session will not be able to exchange data
once the command sent.

6. Prevention
        The only ways known by the writer currently available to prevent
such an attack on a telnet session are the encrypted Kerberos scheme
(application layer) or the TCP crypt implementation
[TCPcrypt] (TCP layer).  Encryption of the data flow prevents any
intrusion or modification of the content. Signature of the data can
also be used. [pgp] is an example of an available way to secure
electronic mail transmission.

7. Morris' Attack Reviewed

        Morris' attack as described in [morris85] assumes that
the attacker can predict the next initial sequence number used by the server
(noted SVR_SEQ_0 in this document) and that the identification scheme is
based on trusted hosts (which means only certain hosts are allowed
to perform some commands on the server without any other identification
process being needed).

        In this attack the cracker initiates the session
by sending a SYN packet to the server using the client (trusted host)
as the source address. The server acknowledge the SYN with a SYN/ACK
packet with SEG_SEQ = SVR_SEQ_0. The attacker then acknowledges that packet
in guessing SVR_SEQ_0. The cracker does not need to sniff the client packets
as long as he can predict SVR_SEQ_0 in order to acknowledge it. This
attack has two main flaws:

 - The client whom the attacker masquerades will receive the SYN/ACK packet
from the server and then could generate a RST packet to the server
since in the client's view no session yet exists. Morris supposes that
one can stop the RST generation by either performing the attack when the
client is down or by overflowing the client's TCP queue so the SYN/ACK

packet will be lost.
 - The attacker cannot receive data from the server. But he can send
data which is sometime enough to compromise a host.

        The are four principal differences between Morris' attack and
the present one:

 - Morris's relies on the trusted hosts identification scheme whereas
the present attack lets the user conduct the identification stage of the
connection.
 - The present attack is a full duplex TCP stream. The attacker can
send and receive data.
 - The present attack uses the ethernet sniffer to predict (or just get)
SVR_SEQ_0.
 - The present attack can be used against any kind of host besides
Unix hosts.

        Morris' attack can easily be extented in regard of the present attack:

 - The sniffer is used to get the server's initial sequence number. Morris'
attack can then be performed against the server. The attacker do not need
to wait for a client to connect.
 - Considering that the client will not send RST packets (for example it is
down) the attacker can establish a full duplex TCP connection
with the server. It can send data and receive data on behalf of the client.
Of course the cracker still has to pass the identification barrier. If the
identification is based on trusted hosts (like NFS or rlogin)
the cracker has full access to the host's services.

        Steven M. Bellovin in [bellovin89] also presents how ICMP
packets can be used to disable one side of the connection. In this case
the attacker gets full control of the session (people have referred
to 'TCP session hijacking'), but this is too easily detected by the user.

8. Conclusion

        Although easy to detect when used on a local network, the attack
presented here is quite efficient on long distance, low bandwidth, high
delay networks (usually WAN). It can be carried with the same resources as
for a passive sniffing attack which have occurred so frequently on the Internet
.
This attack has also the dangerous advantage of being invisible to the user.
While cracking into a host on the Internet is becoming more and more
frequent, the stealthfulness of the attack is now a very important
parameter for the success of the attack and makes it more difficult to
detect.

        When everybody's attention in the Internet is focused on the
emerging new IPv6 protocol to replace the current IPv4, increasing
attacks and the need for secure systems press us to develop and use a
secure transport layer for the Internet community. Options should be
available to send signed and eventually encrypted data  to provide
privacy. And since the signature of the data implies reliability the
signature can be substituted to the current TCP checksum.

        This paper does not attempt to explain all cases
of active attacks using a sniffer. It is more a warning for people using
s/key or Kerberos against the danger of someone sniffing the ethernet.
It provides a few ideas and starting points which can be more deeply studied.
The method presented has been successfully used during our test even with
a very simple attacker's software.

[Bellovin89] "Security Problems in the TCP/IP
        Protocol Suite", Bellovin, S., Computer Communications Review,

        April 1989.

[Kerberos] "Kerberos: An Authentication Service for
        Open Network Systems", Steiner, J., Neuman, C., Schiller, J.,
        USENIX Conference Proceeding, Dallas, Texas, February 1989.

[Morris85] "A Weakness in the 4.2BSD UNIX TCP/IP
        Software", Morris, R., Computing Science Technical Report No 117,
        ATT Bell Laboratories, Murray Hill, New Jersey, 1985.

[PGP] Pretty Good Privacy Version 2.6.1, Philip
        Zimmermann, August 1994.

[RFC 793] Request For Comment 793,
        ``Transmission Control Protocol'', September 1981, J. Postel.

[RFC 854] Request For Comment 854,
        ``Telnet Protocol Specification'', May 1983, J. Postel,
        J. Reynolds

[SKEY] "The S/Key One-time Password System", Haller, N.,
        Proceeding of the Symposium on Network Distributed Systems,
        Security, Internet Society, San Diego, CA, February 1994.

[TCPcrypt] "Public Key Encryption Support for TCP",
        Joncheray, L., Work in progress, May 1995.

[TCPDUMP] tcpdump(8) Version 2.2.1, Van Jacobson,
        Craig Leres, Steven Berkeley, University of California, Berkeley, CA.