

Steffen Wendzel

vstt [Very Strange Tunneling Tool] documentation

Contents

1 Disclaimer	3
2 Introduction	3
3 How to use it?	3
4 Examples	4
4.1 example 1 (without a tcp connection)	4
4.2 example 2 (tunneling a ssh connection)	5
5 Protocols	6
5.1 none	6
5.2 POP3	6
5.3 ICMP	6
6 Comments, Feedback	6

1 Disclaimer

This tool is for legal purposes only! Read the LICENSE file for license details.

2 Introduction

vsst is a tunneling tool (primary for TCP connections). It can send your data via different protocols. Send me patches, if you port it to new systems or if you fixed a bug ;-)

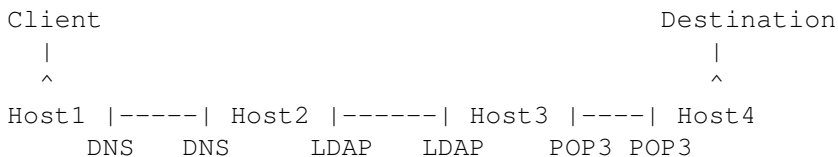
currently tested systems are:

- Linux 2.6.x (i386 or amd64)
- OpenBSD 3.x to 4.0-current (i386 and amd64)
- SHOULD work too: FreeBSD, NetBSD and Solaris (need a Makefile modification)

vsst can tunnel your data via the following protocols:

- NONE (a pseudo protocol) - 99% done
- ICMP - 95% done
- POP3 - 90% done
- DNS - 10% done
- SUNRPC - 0% done
- LDAP - 0% done

You can use different protocol connections between vsst hosts. here is an example net using 3 different vsst tunnels:



Host2 and Host3 are just vsst gateways.

3 How to use it?

A: With local FIFOs or with TCP sockets you can connect to.

It's very simple to create tunnels with vsst: it works locally with pipes. You can send data into this pipes and you can read data from this pipes.

vsst has 2 binaries (to make it possible to create gateways) that make use of the following fifos in /tmp:

binary name	input fifo	output fifo
vsst	/tmp/.vsst_send2peer	/tmp/.vsst_recvfppeer
vsst2	/tmp/.vsst_send2peer2	/tmp/.vsst_recvfppeer2

You can send data into the connection via writing data into the input fifo and you can read received data from the peer via reading from the output fifo.

Q: But aehmmm.... I want to use sockets because my TCP app (telnet or ssh for example) uses TCP and not fifos.

A: No problem: you have to use the s2f tool included in the code what bindes a TCP socket to a fifo.

4 Examples

Note: vstt normally produces one 'connection refused' error a second if the other peer isn't already started. You don't need to take care about that.

4.1 example 1 (without a tcp connection)

Okay, let's create a simple POP3 tunnel using vstt on localhost. We want to send a file through the tunnel and read it with the shipped tool 'reader'.

This needs different parameters to start vstt:

```
-p pop3      <- set the protocol to pop3
-r n         <- receive data on port n
-t m         <- send data to the peer at port m
-a 127.0.0.1 <- the address of the peer
```

Okay. For a tunnel we need 2 peers. In our case they are both on our local machine. This is why we need both binaries because we need 4 fifos.

First, we start vstt in one xterm and then we start vstt2 on another one:

```
one\$ ./vstt -p pop3 -r 10001 -t 10002 -a 127.0.0.1
init_pop3();
fork_childs();
connecting to peer ...
server: waiting for connection...connect(): Connection refused
connecting to peer ...
con establ
client: waiting for data from fifo...
connection established
waiting for data...

two\$ ./vstt2 -p pop3 -r 10002 -t 10001 -a 127.0.0.1
init_pop3();
fork_childs();
connecting to peer ...
server: waiting for connection...connection established
waiting for data...
con establ
client: waiting for data from fifo...
```

Okay, our tunnel is now established. Let's play around with it. In the source code is a tool called 'reader' included, what reads our data from a file we give it as parameter. In this case we use it to read the data from the 2nd output fifo:

```
\$ ./reader /tmp/.vstt_recvfppeer2
```

Okay, we now have a tunnel and a tool that prints all data we send from the first binary to the second one. To test it all, we now write data into the sending fifo of the first binary:

```
\$ cat /etc/resolv.conf > .vstt_send2peer
```

If we now look at the output of the ./reader tool, we will see the content of /etc/resolv.conf that was tunneled via pop3 between vstt and vstt2.

4.2 example 2 (tunneling a ssh connection)

Now we want to tunnel a SSH connection between two hosts over port 80 (because our firewall doesn't block it or so). We use the protocol 'none' because it's fast, works very well and because that SSH already is encrypted and looks binary enough to look like a valid HTTP download ;-)

Note: You need root access to bind ports less than 1024 under unix(-like) systems.

This works as follows: Both systems start a vstt-tunnel they can communicate with. On the SSH-server we connect our vstt-Fifo with the SSH service on port 22 (what can be done by using the s2f tool – very simple).

On the Client machine we use the tool s2f too (but in server mode). s2f communicates with the local vstt endpoint via it's Fifo. And then we connect to the s2f port using our local ssh client. That's all.

Okay, Let's start.

Say that eygo (192.168.2.20) is the machine with the SSH-Server and that hikoki (192.168.2.21) is the server with the SSH client.

On the first terminal (xterm or a console terminal or whatever), we start the vstt tunnel software. We receive data on port 80 and send data to port 80 at the other vstt-endpoint.

```
eygo\# ./vstt -p none -r 80 -t 80 -a 192.168.2.20
client: connecting to peer ...
server: waiting for connection...
none(or pop3 and so on)_client: connect(): Connection refused
none(or pop3 and so on)_client: connect(): Connection refused
none(or pop3 and so on)_client: connect(): Connection refused
none(or pop3 and so on)_client: connect(): Connection refused
none(or pop3 and so on)_client: connect(): Connection refused
...
...
```

On the 2nd terminal we start s2f. It will listen on port 10003. We will connect to this port with the ssh client if the tunnel works.

```
eygo\# ./s2f -s -p 10003
```

IMPORTANT NOTE: If you don't want to start s2f by hand, you can also let vstt do that by using -c <port> [-s] parameters! Instead of starting vstt+s2f, you could start only vstt in this example:

```
# vstt -p none -r 80 -t 80 -a 192.168.2.20 -c 10003 -s
```

btw: -s means to run as a server and to use the port given with -p as the listen port instead as the port to connect to.

On eygo, we start vstt too:

```
eygo\# ./vstt -p none -r 80 -t 80 -a 192.168.2.21
client: connecting to peer ...
server: waiting for connection...
wrapper_tcpserver: connection established => waiting for data...
==> con establ
client: waiting for data from fifo...
```

And we connect the vstt-Fifos to the local SSH-Server running on Port 22 by s2f:

```
eygo\# ./s2f -p 22  
connected.
```

IMPORTANT NOTE: You could alternatively only start vstt one time without calling s2f:

```
# ./vstt -p none -r 80 -t 80 -a 192.168.2.21 -c 22
```

And now, you can connect with ssh to the localhost port 10003 on the first machine (hikoki).

```
hikoki\$ ssh user@127.0.0.1 -p 10003
```

5 Protocols

5.1 none

The non protocol is used for a blank tunnel. For example: You sit behind a firewall that only let's you use port 80 but you want to connect to your IRC-server at home. You can use the none protocol to redirect the connection over port 80 to bypass the firewall and enjoy your IRC session.

5.2 POP3

This is a little bit more advanced. A POP3 tunnel is slow but it can hide your data in RETR-requests. If you want to hide your data a little bit: use POP3 (or ICMP).

5.3 ICMP

If all TCP+UDP ports are blocked, an ICMP tunnel can work anyway. vstt sends your data as payload in ICMP echo datagrams. vstt can re-send lost packets, re-calculates the checksum to be sure that no problems will happen and can also send big packets from your applications within many small ICMP packets that will be re-assembled by the peer.

Future versions will include an encryption algorithm for ICMP.

6 Comments, Feedback

Please send me feedback, typos, bug reports and implementation wishes to my eMail address `cdp (at) doomed-reality (dot) org` to make vstt better.