sshuttle documentation

Release 0.77.2.dev0+ng8bdefcd.d20160306

Brian May

March 06, 2016

Contents

1 Overview **3**

2 Requirements **5**
- 2.1 Client side Requirements . . . 5
- 2.2 Server side Requirements . . . 6
- 2.3 Additional Suggested Software . . . 6

3 Installation **7**

4 Usage **9**
- 4.1 Usage Notes . . . 9

5 Platform Specific Notes **11**
- 5.1 TPROXY . . . 11
- 5.2 Microsoft Windows . . . 12

6 sshuttle **13**
- 6.1 Synopsis . . . 13
- 6.2 Description . . . 13
- 6.3 Options . . . 13
- 6.4 Examples . . . 15
- 6.5 Discussion . . . 16

7 How it works **17**

8 Support **19**

9 Useless Trivia **21**

10 Changelog **23**
- 10.1 Release 0.77.1 (Mar 7, 2016) . . . 23
- 10.2 Release 0.77 (Mar 3, 2016) . . . 23
- 10.3 Release 0.76 (Jan 17, 2016) . . . 23
- 10.4 Release 0.75 (Jan 12, 2016) . . . 23
- 10.5 Release 0.74 (Jan 10, 2016) . . . 23

11 Indices and tables **25**

Date March 06, 2016

Version 0.77.2.dev0+ng8bdefcd.d20160306

Contents:

CHAPTER 1

Overview

As far as I know, sshuttle is the only program that solves the following common case:

- Your client machine (or router) is Linux, FreeBSD, or MacOS.
- You have access to a remote network via ssh.
- You don't necessarily have admin access on the remote network.
- The remote network has no VPN, or only stupid/complex VPN protocols (IPsec, PPTP, etc). Or maybe you *are* the admin and you just got frustrated with the awful state of VPN tools.
- You don't want to create an ssh port forward for every single host/port on the remote network.
- You hate openssh's port forwarding because it's randomly slow and/or stupid.
- You can't use openssh's PermitTunnel feature because it's disabled by default on openssh servers; plus it does TCP-over-TCP, which has terrible performance (see below).

CHAPTER 2

Requirements

2.1 Client side Requirements

- sudo, or root access on your client machine. (The server doesn't need admin access.)
- Python 2.7 or Python 3.5.

2.1.1 Linux with NAT method

Supports:

- IPv4 TCP
- IPv4 DNS

Requires:

- iptables DNAT, REDIRECT, and ttl modules.

2.1.2 Linux with TPROXY method

Supports:

- IPv4 TCP
- IPv4 UDP (requires `recmsg` - see below)
- IPv6 DNS (requires `recmsg` - see below)
- IPv6 TCP
- IPv6 UDP (requires `recmsg` - see below)
- IPv6 DNS (requires `recmsg` - see below)

Full UDP or DNS support with the TPROXY method requires the `recvmsg()` syscall. This is not available in Python 2, however is in Python 3.5 and later. Under Python 2 you might find it sufficient installing PyXAPI to get the `recvmsg()` function. See TPROXY for more information.

2.1.3 MacOS / FreeBSD / OpenBSD

Method: pf

Supports:

- IPv4 TCP
- IPv4 DNS

Requires:

- You need to have the pfctl command.

2.1.4 Windows

Not officially supported, however can be made to work with Vagrant. Requires cmd.exe with Administrator access. See Microsoft Windows for more information.

2.2 Server side Requirements

Python 2.7 or Python 3.5.

2.3 Additional Suggested Software

- You may want to use autossh, available in various package management systems

CHAPTER 3

Installation

- From PyPI:

```
pip install sshuttle
```

- Clone:

```
git clone https://github.com/sshuttle/sshuttle.git
./setup.py install
```

CHAPTER 4

Usage

Note: For information on usage with Windows, see the Microsoft Windows section. For information on using the TProxy method, see the TPROXY section.

Forward all traffic:

```
sshuttle -r username@sshserver 0.0.0.0/0
```

- Use the *sshuttle -r* parameter to specify a remote server.
- By default sshuttle will automatically choose a method to use. Override with the *sshuttle --method* parameter.
- There is a shortcut for 0.0.0.0/0 for those that value their wrists:

```
    sshuttle -r username@sshserver 0/0
```

If you would also like your DNS queries to be proxied through the DNS server of the server you are connect to:

```
sshuttle --dns -r username@sshserver 0/0
```

The above is probably what you want to use to prevent local network attacks such as Firesheep and friends. See the documentation for the *sshuttle --dns* parameter.

(You may be prompted for one or more passwords; first, the local password to become root using sudo, and then the remote ssh password. Or you might have sudo and ssh set up to not require passwords, in which case you won't be prompted at all.)

4.1 Usage Notes

That's it! Now your local machine can access the remote network as if you were right there. And if your "client" machine is a router, everyone on your local network can make connections to your remote network.

You don't need to install sshuttle on the remote server; the remote server just needs to have python available. sshuttle will automatically upload and run its source code to the remote python interpreter.

This creates a transparent proxy server on your local machine for all IP addresses that match 0.0.0.0/0. (You can use more specific IP addresses if you want; use any number of IP addresses or subnets to change which addresses get proxied. Using 0.0.0.0/0 proxies *everything*, which is interesting if you don't trust the people on your local network.)

Any TCP session you initiate to one of the proxied IP addresses will be captured by sshuttle and sent over an ssh session to the remote copy of sshuttle, which will then regenerate the connection on that end, and funnel the data back and forth through ssh.

Fun, right? A poor man's instant VPN, and you don't even have to have admin access on the server.

CHAPTER 5

Platform Specific Notes

Contents:

5.1 TPROXY

TPROXY is the only method that has full support of IPv6 and UDP.

There are some things you need to consider for TPROXY to work:

- The following commands need to be run first as root. This only needs to be done once after booting up:

```
ip route add local default dev lo table 100
ip rule add fwmark 1 lookup 100
ip -6 route add local default dev lo table 100
ip -6 rule add fwmark 1 lookup 100
```

- The `--auto-nets` feature does not detect IPv6 routes automatically. Add IPv6 routes manually. e.g. by adding `'::/0'` to the end of the command line.
- The client needs to be run as root. e.g.:

```
sudo SSH_AUTH_SOCK="$SSH_AUTH_SOCK" $HOME/tree/sshuttle.tproxy/sshuttle  --method=tproxy ...
```

- You may need to exclude the IP address of the server you are connecting to. Otherwise sshuttle may attempt to intercept the ssh packets, which will not work. Use the `--exclude` parameter for this.
- Similarly, UDP return packets (including DNS) could get intercepted and bounced back. This is the case if you have a broad subnet such as `0.0.0.0/0` or `::/0` that includes the IP address of the client. Use the `--exclude` parameter for this.
- You need the `--method=tproxy` parameter, as above.
- The routes for the outgoing packets must already exist. For example, if your connection does not have IPv6 support, no IPv6 routes will exist, IPv6 packets will not be generated and sshuttle cannot intercept them:

```
telnet -6 www.google.com 80
Trying 2404:6800:4001:805::1010...
telnet: Unable to connect to remote host: Network is unreachable
```

  Add some dummy routes to external interfaces. Make sure they get removed however after sshuttle exits.

5.2 Microsoft Windows

Currently there is no built in support for running sshuttle directly on Microsoft Windows.

What we can really do is to create a Linux VM with Vagrant (or simply Virtualbox if you like). In the Vagrant settings, remember to turn on bridged NIC. Then, run sshuttle inside the VM like below:

```
sshuttle -l 0.0.0.0 -x 10.0.0.0/8 -x 192.168.0.0/16 0/0
```

10.0.0.0/8 excludes NAT traffic of Vagrant and 192.168.0.0/16 excludes traffic to local area network (assuming that we're using 192.168.0.0 subnet).

Assuming the VM has the IP 192.168.1.200 obtained on the bridge NIC (we can configure that in Vagrant), we can then ask Windows to route all its traffic via the VM by running the following in cmd.exe with admin right:

```
route add 0.0.0.0 mask 0.0.0.0 192.168.1.200
```

CHAPTER 6

sshuttle

6.1 Synopsis

sshuttle [*options*] [**-r** *[username@]sshserver[:port]*] <*subnets ...*>

6.2 Description

sshuttle allows you to create a VPN connection from your machine to any remote server that you can connect to via ssh, as long as that server has python 2.3 or higher.

To work, you must have root access on the local machine, but you can have a normal account on the server.

It's valid to run **sshuttle** more than once simultaneously on a single client machine, connecting to a different server every time, so you can be on more than one VPN at once.

If run on a router, **sshuttle** can forward traffic for your entire subnet to the VPN.

6.3 Options

subnets
: A list of subnets to route over the VPN, in the form `a.b.c.d[/width]`. Valid examples are 1.2.3.4 (a single IP address), 1.2.3.4/32 (equivalent to 1.2.3.4), 1.2.3.0/24 (a 24-bit subnet, ie. with a 255.255.255.0 netmask), and 0/0 ('just route everything through the VPN').

--method `[auto|nat|tproxy|pf]`
: Which firewall method should sshuttle use? For auto, sshuttle attempts to guess the appropriate method depending on what it can find in PATH. The default value is auto.

-l, --listen`=[ip:]port`
: Use this ip address and port number as the transparent proxy port. By default **sshuttle** finds an available port automatically and listens on IP 127.0.0.1 (localhost), so you don't need to override it, and connections are only proxied from the local machine, not from outside machines. If you want to accept connections from other machines on your network (ie. to run **sshuttle** on a router) try enabling IP Forwarding in your kernel, then using `--listen 0.0.0.0:0`.

  For the tproxy method this can be an IPv6 address. Use this option twice if required, to provide both IPv4 and IPv6 addresses.

-H, --auto-hosts
Scan for remote hostnames and update the local /etc/hosts file with matching entries for as long as the VPN is open. This is nicer than changing your system's DNS (/etc/resolv.conf) settings, for several reasons. First, hostnames are added without domain names attached, so you can `ssh thatserver` without worrying if your local domain matches the remote one. Second, if you **sshuttle** into more than one VPN at a time, it's impossible to use more than one DNS server at once anyway, but **sshuttle** correctly merges /etc/hosts entries between all running copies. Third, if you're only routing a few subnets over the VPN, you probably would prefer to keep using your local DNS server for everything else.

-N, --auto-nets
In addition to the subnets provided on the command line, ask the server which subnets it thinks we should route, and route those automatically. The suggestions are taken automatically from the server's routing table.

--dns
Capture local DNS requests and forward to the remote DNS server.

--python
Specify the name/path of the remote python interpreter. The default is just `python`, which means to use the default python interpreter on the remote system's PATH.

-r, --remote=`[username@]sshserver[:port]`
The remote hostname and optional username and ssh port number to use for connecting to the remote server. For example, example.com, testuser@example.com, testuser@example.com:2222, or example.com:2244.

-x, --exclude=`subnet`
Explicitly exclude this subnet from forwarding. The format of this option is the same as the `<subnets>` option. To exclude more than one subnet, specify the `-x` option more than once. You can say something like `0/0 -x 1.2.3.0/24` to forward everything except the local subnet over the VPN, for example.

-X, --exclude-from=`file`
Exclude the subnets specified in a file, one subnet per line. Useful when you have lots of subnets to exclude.

-v, --verbose
Print more information about the session. This option can be used more than once for increased verbosity. By default, **sshuttle** prints only error messages.

-e, --ssh-cmd
The command to use to connect to the remote server. The default is just `ssh`. Use this if your ssh client is in a non-standard location or you want to provide extra options to the ssh command, for example, `-e 'ssh -v'`.

--seed-hosts
A comma-separated list of hostnames to use to initialize the *--auto-hosts* scan algorithm. *--auto-hosts* does things like poll local SMB servers for lists of local hostnames, but can speed things up if you use this option to give it a few names to start from.

--no-latency-control
Sacrifice latency to improve bandwidth benchmarks. ssh uses really big socket buffers, which can overload the connection if you start doing large file transfers, thus making all your other sessions inside the same tunnel go slowly. Normally, **sshuttle** tries to avoid this problem using a "fullness check" that allows only a certain amount of outstanding data to be buffered at a time. But on high-bandwidth links, this can leave a lot of your bandwidth underutilized. It also makes **sshuttle** seem slow in bandwidth benchmarks (benchmarks rarely test ping latency, which is what **sshuttle** is trying to control). This option disables the latency control feature, maximizing bandwidth usage. Use at your own risk.

-D, --daemon
Automatically fork into the background after connecting to the remote server. Implies *--syslog*.

--syslog
after connecting, send all log messages to the *syslog(3)* service instead of stderr. This is implicit if you use *--daemon*.

--pidfile=pidfilename
: when using *--daemon*, save **sshuttle**'s pid to *pidfilename*. The default is `sshuttle.pid` in the current directory.

--disable-ipv6
: If using the tproxy method, this will disable IPv6 support.

--firewall
: (internal use only) run the firewall manager. This is the only part of **sshuttle** that must run as root. If you start **sshuttle** as a non-root user, it will automatically run `sudo` or `su` to start the firewall manager, but the core of **sshuttle** still runs as a normal user.

--hostwatch
: (internal use only) run the hostwatch daemon. This process runs on the server side and collects hostnames for the *--auto-hosts* option. Using this option by itself makes it a lot easier to debug and test the *--auto-hosts* feature.

6.4 Examples

Test locally by proxying all local connections, without using ssh:

```
$ sshuttle -v 0/0

Starting sshuttle proxy.
Listening on ('0.0.0.0', 12300).
[local sudo] Password:
firewall manager ready.
c : connecting to server...
 s: available routes:
 s:   192.168.42.0/24
c : connected.
firewall manager: starting transproxy.
c : Accept: 192.168.42.106:50035 -> 192.168.42.121:139.
c : Accept: 192.168.42.121:47523 -> 77.141.99.22:443.
    ...etc...
^C
firewall manager: undoing changes.
KeyboardInterrupt
c : Keyboard interrupt: exiting.
c : SW#8:192.168.42.121:47523: deleting
c : SW#6:192.168.42.106:50035: deleting
```

Test connection to a remote server, with automatic hostname and subnet guessing:

```
$ sshuttle -vNHr example.org

Starting sshuttle proxy.
Listening on ('0.0.0.0', 12300).
firewall manager ready.
c : connecting to server...
 s: available routes:
 s:   77.141.99.0/24
c : connected.
c : seed_hosts: []
firewall manager: starting transproxy.
hostwatch: Found: testbox1: 1.2.3.4
hostwatch: Found: mytest2: 5.6.7.8
```

```
hostwatch: Found: domaincontroller: 99.1.2.3
c : Accept: 192.168.42.121:60554 -> 77.141.99.22:22.
^C
firewall manager: undoing changes.
c : Keyboard interrupt: exiting.
c : SW#6:192.168.42.121:60554: deleting
```

6.5 Discussion

When it starts, **sshuttle** creates an ssh session to the server specified by the `-r` option. If `-r` is omitted, it will start both its client and server locally, which is sometimes useful for testing.

After connecting to the remote server, **sshuttle** uploads its (python) source code to the remote end and executes it there. Thus, you don't need to install **sshuttle** on the remote server, and there are never **sshuttle** version conflicts between client and server.

Unlike most VPNs, **sshuttle** forwards sessions, not packets. That is, it uses kernel transparent proxying (*iptables REDIRECT* rules on Linux) to capture outgoing TCP sessions, then creates entirely separate TCP sessions out to the original destination at the other end of the tunnel.

Packet-level forwarding (eg. using the tun/tap devices on Linux) seems elegant at first, but it results in several problems, notably the 'tcp over tcp' problem. The tcp protocol depends fundamentally on packets being dropped in order to implement its congestion control agorithm; if you pass tcp packets through a tcp-based tunnel (such as ssh), the inner tcp packets will never be dropped, and so the inner tcp stream's congestion control will be completely broken, and performance will be terrible. Thus, packet-based VPNs (such as IPsec and openvpn) cannot use tcp-based encrypted streams like ssh or ssl, and have to implement their own encryption from scratch, which is very complex and error prone.

sshuttle's simplicity comes from the fact that it can safely use the existing ssh encrypted tunnel without incurring a performance penalty. It does this by letting the client-side kernel manage the incoming tcp stream, and the server-side kernel manage the outgoing tcp stream; there is no need for congestion control to be shared between the two separate streams, so a tcp-based tunnel is fine.

See also:

ssh(1), *python(1)*

CHAPTER 7

How it works

sshuttle is not exactly a VPN, and not exactly port forwarding. It's kind of both, and kind of neither.

It's like a VPN, since it can forward every port on an entire network, not just ports you specify. Conveniently, it lets you use the "real" IP addresses of each host rather than faking port numbers on localhost.

On the other hand, the way it *works* is more like ssh port forwarding than a VPN. Normally, a VPN forwards your data one packet at a time, and doesn't care about individual connections; ie. it's "stateless" with respect to the traffic. sshuttle is the opposite of stateless; it tracks every single connection.

You could compare sshuttle to something like the old Slirp program, which was a userspace TCP/IP implementation that did something similar. But it operated on a packet-by-packet basis on the client side, reassembling the packets on the server side. That worked okay back in the "real live serial port" days, because serial ports had predictable latency and buffering.

But you can't safely just forward TCP packets over a TCP session (like ssh), because TCP's performance depends fundamentally on packet loss; it *must* experience packet loss in order to know when to slow down! At the same time, the outer TCP session (ssh, in this case) is a reliable transport, which means that what you forward through the tunnel *never* experiences packet loss. The ssh session itself experiences packet loss, of course, but TCP fixes it up and ssh (and thus you) never know the difference. But neither does your inner TCP session, and extremely screwy performance ensues.

sshuttle assembles the TCP stream locally, multiplexes it statefully over an ssh session, and disassembles it back into packets at the other end. So it never ends up doing TCP-over-TCP. It's just data-over-TCP, which is safe.

CHAPTER 8

Support

Mailing list:

- Subscribe by sending a message to <sshuttle+subscribe@googlegroups.com>
- List archives are at: http://groups.google.com/group/sshuttle

Issue tracker and pull requests at github:

- https://github.com/sshuttle/sshuttle

CHAPTER 9

Useless Trivia

This section written by the original author, Avery Pennarun <apenwarr@gmail.com>.

Back in 1998, I released the first version of Tunnel Vision, a semi-intelligent VPN client for Linux. Unfortunately, I made two big mistakes: I implemented the key exchange myself (oops), and I ended up doing TCP-over-TCP (double oops). The resulting program worked okay - and people used it for years - but the performance was always a bit funny. And nobody ever found any security flaws in my key exchange, either, but that doesn't mean anything. :)

The same year, dcoombs and I also released Fast Forward, a proxy server supporting transparent proxying. Among other things, we used it for automatically splitting traffic across more than one Internet connection (a tool we called "Double Vision").

I was still in university at the time. A couple years after that, one of my professors was working with some graduate students on the technology that would eventually become Slipstream Internet Acceleration. He asked me to do a contract for him to build an initial prototype of a transparent proxy server for mobile networks. The idea was similar to sshuttle: if you reassemble and then disassemble the TCP packets, you can reduce latency and improve performance vs. just forwarding the packets over a plain VPN or mobile network. (It's unlikely that any of my code has persisted in the Slipstream product today, but the concept is still pretty cool. I'm still horrified that people use plain TCP on complex mobile networks with crazily variable latency, for which it was never really intended.)

That project I did for Slipstream was what first gave me the idea to merge the concepts of Fast Forward, Double Vision, and Tunnel Vision into a single program that was the best of all worlds. And here we are, at last. You're welcome.

CHAPTER 10

Changelog

10.1 Release 0.77.1 (Mar 7, 2016)

- Use semantic versioning. http://semver.org/
- Update GPL 2 license text.
- New release to fix PyPI.

10.2 Release 0.77 (Mar 3, 2016)

- Various bug fixes.
- Fix Documentation.
- Add fix for MacOS X issue.
- Add support for OpenBSD.

10.3 Release 0.76 (Jan 17, 2016)

- Add option to disable IPv6 support.
- Update documentation.
- Move documentation, including man page, to Sphinx.
- Use setuptools-scm for automatic versioning.

10.4 Release 0.75 (Jan 12, 2016)

- Revert change that broke sshuttle entry point.

10.5 Release 0.74 (Jan 10, 2016)

- Add CHANGES.rst file.

- Numerous bug fixes.
- Python 3.5 fixes.
- PF fixes, especially for BSD.

CHAPTER 11

Indices and tables

- genindex
- search

Symbols

–disable-ipv6
    sshuttle command line option, 15
–dns
    sshuttle command line option, 14
–firewall
    sshuttle command line option, 15
–hostwatch
    sshuttle command line option, 15
–method [auto|nat|tproxy|pf]
    sshuttle command line option, 13
–no-latency-control
    sshuttle command line option, 14
–pidfile=pidfilename
    sshuttle command line option, 14
–python
    sshuttle command line option, 14
–seed-hosts
    sshuttle command line option, 14
–syslog
    sshuttle command line option, 14
-D, –daemon
    sshuttle command line option, 14
-H, –auto-hosts
    sshuttle command line option, 13
-N, –auto-nets
    sshuttle command line option, 14
-X, –exclude-from=file
    sshuttle command line option, 14
-e, –ssh-cmd
    sshuttle command line option, 14
-l, –listen=[ip:]port
    sshuttle command line option, 13
]sshserver[:port]
    sshuttle command line option, 14
-v, –verbose
    sshuttle command line option, 14
-x, –exclude=subnet
    sshuttle command line option, 14

S

sshuttle command line option
    –disable-ipv6, 15
    –dns, 14
    –firewall, 15
    –hostwatch, 15
    –method [auto|nat|tproxy|pf], 13
    –no-latency-control, 14
    –pidfile=pidfilename, 14
    –python, 14
    –seed-hosts, 14
    –syslog, 14
    -D, –daemon, 14
    -H, –auto-hosts, 13
    -N, –auto-nets, 14
    -X, –exclude-from=file, 14
    -e, –ssh-cmd, 14
    -l, –listen=[ip:]port, 13
    ]sshserver[:port], 14
    -v, –verbose, 14
    -x, –exclude=subnet, 14
    subnets, 13
subnets
    sshuttle command line option, 13