Using TCP Wrappers to control access

Skill Level: Introductory

David Tansley
System Administrator
Ace Europe

12 Jul 2011

TCP Wrappers allows system administrators to control and log incoming TCP-based connections to the local host run from inetd.conf. TCP wrappers, often called wrappers, can lock down popular TCP inbound clients on your AIX box quickly. Find out how wrappers can easily protect and secure your machines.

Introduction

TCP wrappers, often called just wrappers, is written by Wieste Venema and has been around for quite a few years. The idea behind it is simple, but the effect is that popular TCP inbound clients can be locked down on your AIX (UNIX®/Linux®) box quickly and easily.

Wrappers allow system administrators to control access of TCP-based services or daemons that are wrappers aware. Tcpd controls TCP daemons that are run from /etc/inetd.conf. However, many TCP-based applications have been compiled with wrappers support (usually using the libwrap library) and are wrapper aware; it does not matter that they are not controlled from /etc/inetd.conf. Some common candidates that are used under wrappers for access control are telnet, ssh, sendmail, ftp packages, pop3, and stunnel.

Wrappers do offer limited support for UDP-based connections, but I suggest using your built-in or third party firewall for UDP based access

To see if your application has wrapper support, use the strings command and grep for hosts_access or host:

```
# strings /usr/sbin/sshd|grep hosts_access
```

```
@(#)65  1.1  src/tcpwrapper/usr/sbin/tcpwrapper/hosts_access.c, tcpwrap, 53twrp2
10, 0617A_53twrp210 2/27/06 04:52:25
```

Or, you can use the ldd command:

```
ldd </path/application> | grep libwrap
```

The wrappers daemon is called tcpd. It is called instead of the real daemon in the /etc/inetd.conf file. Tcpd reads two files, hosts.allow and hosts.deny, based on the rules in these files. When the first rule match is found, the calling client is either denied or allowed access. All actions are logged to the messages file or to a specified log file that can be determined via syslog. Traditionally, hosts.allow contains the allowed access rules and hosts.deny contains the denied rules.

Looking at a typical telnet client scenario, this is how it works. The client tries to connect via a telnet session to a host that has wrappers configured. The client unknown to them connects to the wrapper daemon instead of the real telnetd daemon. This client is either allowed or denied access based on the rules contained in the hosts.allow or hosts.deny file. If denied, the telnet connection is terminated. If the client is allowed access based on the allowed rules, then tcpd hands over control to the real daemon that was called (in this case, telnetd). In either case, a grant or denied connection is logged via syslog.

In this article, I will demonstrate how to configure wrappers from source and discuss and review rules on blocking or allowing different remote service connections.

Building wrappers

Wrappers can be downloaded for either IPv4 or IPv6 support. The version to download is 7.6. In my experience of administering over 50 AIX boxes, the IPv6 version runs quite happily on a IPv4 only connection, so use this version. To build wrappers, you need a C compiler; I am using gcc from the AIX toolbox. Details of all downloads can be found in the Resources section.

```
# gcc -v
Target: powerpc-ibm-aix5.3.0.0
...
gcc version 4.2.0
```

Once downloaded, unzip and untar the wrappers. The files are extracted into the tcp_wrappers_7_6-ipv6.4 directory:

```
# gunzip tcp_wrappers_7.6-ipv6.4.tar.gz
# tar -xvf tcp_wrappers_7.6-ipv6.4.tar
x tcp_wrappers_7.6-ipv6.4
```

```
x tcp_wrappers_7.6-ipv6.4/DISCLAIMER, 792 bytes, 2 media blocks.
x tcp_wrappers_7.6-ipv6.4/BLURB, 1736 bytes, 4 media blocks.
x tcp_wrappers_7.6-ipv6.4/CHANGES, 19195 bytes, 38 media blocks.
x tcp_wrappers_7.6-ipv6.4/Makefile, 32976 bytes, 65 media blocks.
….
#
```

The next task is to customise wrappers to allow rule checking, lookups and lookup verification (if required), logging, spawning programs and the use of banners.

In the makefile, the following entries are enabled by uncommenting the lines, as shown below:

```
# SysV.4 Solaris 2.x OSF AIX
REAL_DAEMON_DIR=/usr/sbin
```

The real_daemon informs wrappers where the real daemons live like ftpd, telnetd, and sshd.

```
HOSTNAME= -DALWAYS_HOSTNAME
```

Always attempt a hostname lookup:

```
ACCESS  = -DHOSTS_ACCESS
```

Enable host control access is the whole point of wrappers, so make sure it is uncommented.

```
STYLE   = -DPROCESS_OPTIONS     # Enable language extensions.
```

Enable language extensions, for banner messages, and using shell commands with spawn and twist.

```
FACILITY= LOG_MAIL
```

Make sure wrappers knows about the syslog mechanism:

```
FACILITY=LOG_LOCAL0
```

If you prefer, you can send messages from wrappers to a specific log file instead using LOCAL:

```
IPV6 = -DHAVE_IPV6
```

This allows IPv6.

If you are running IPv4 on your hosts, still uncomment the following or the client's IP address may not be resolvable under wrappers. Typically it could log an IP as 0.0.0.0. instead of the real IP.

```
SEVERITY= LOG_INFO
```

The logging level via syslog:

```
PARANOID= -DPARANOID
```

Wrappers do a forward and reverse lookup thus checking the IP against the host-name. Action is taken if this condition is not met, as denoted in the access rules. This can cause issues with connecting clients if the DNS is not correctly set-up. If this could be an issue with your security policy, then do not uncomment this line.

In the actual AIX build segment, we have:

```
generic aix osf alpha dynix:
        @make REAL_DAEMON_DIR=$(REAL_DAEMON_DIR) STYLE=$(STYLE) \
        LIBS= RANLIB=ranlib ARFLAGS=rv AUX_OBJ=setenv.o \
        NETGROUP=-DNETGROUP TLI= IPV6="$(IPV6)" all
```

Save and exit the changes; then build it.

```
# make CC=gcc aix
```

Once completed, the following binaries are compiled:

tcpd	The tcpd binary is launched instead of the real daemon that services the client from /etc/inetd.conf. If a third party application has been built with wrappers support, that service or daemon consults the hosts.allow or hosts.deny file where access is either be granted or denied, as decided by that third party daemon or service.
tcpdmatch	This utility can help in testing and diagnosing how tcpd deals with an incoming request. This utility is quite useful in that you can test the rules for access control by supplying a daemon name

	and client.
tcpdchk	This utility reads the current hosts.allow and host.deny file if present and checks for any syntax errors.
try-from	This utility helps in troubleshooting access rules, it is executed from a remote shell to verify if the hostname and IP are returned.
safe_finger	This utility wraps around the finger command to determine reverse lookups on a client.
libwrap.a	This library can be used when building other applications to provide wrapper support. If compiled with libwrap, the third party application should honor the hosts.allow and hosts.deny rules.

The next task is to copy those binaries and library into your local bin and sbin directory. In this demonstration, I will copy the files into the following directories:

```
# cp tcpd /usr/sbin
# cp tcpdmatch /usr/local/bin
# cp tcpdchk /usr/local/bin
# cp try-from /usr/local/bin
# cp safe_finger /usr/local/bin
```

And finally the libwrap:

```
# cp libwrap.a /usr/local/lib
```

Be sure to check that the permissions are executable for the tcpd binaries.

Configuring wrappers

The next task is to invoke tcpd instead of the real daemon or service from /etc/inetd.conf. First, create a backup of /etc/inetd.conf. In this demonstration, I am only going to replace daemons I wish to control access. I am going to replace ftpd and telnetd with tcpd as the calling daemon from /etc/inetd.conf. Feel free to protect other daemons in this file. The /etc/inetd.conf lines we are interested in are:

```
ftp     stream  tcp6      nowait  root    /usr/sbin/ftpd              ftpd
telnet  stream  tcp6    nowait  root    /usr/sbin/telnetd       telnetd -a
```

We simply replace the daemons that we wish to protect, in this case ftpd and telnetd with tcpd. So, after editing we would have the following configuration change:

```
ftp     stream  tcp6    nowait  root    /usr/sbin//tcpd         ftpd
telnet  stream  tcp6    nowait  root    /usr/sbin/tcpd       telnetd -a
```

Next, refresh inetd for the changes to take effect.

```
# refresh -s inetd
```

As discussed earlier, for other wrapper aware built third party applications, the hosts.allow and hosts.deny files are consulted. It is that application's responsibility to honor the rules in the files, thus deny or allow access to its own clients. Let's now configure the hosts files.

Logging of wrappers log by default to /var/adm/messages, using the facility level of authority:

```
auth.info    /var/adm/messages
```

If you have specified the LOCAL facility in the makefile, then you must tell syslog what file to use in syslog.conf. In the following example, all messages using LOCAL from wrappers is logged to the /var/adm/wrappers.log file.

```
local0.info             /var/adm/wrappers.log
```

Be sure to create the file wrappers.log before restarting syslog:

```
# refresh -s syslogd
```

The rules

To turn wrappers off, simply mv the hosts.allow and hosts.deny file to different file-names. If there are no allow or deny files present, wrappers will not use access control, effectively turning wrappers off. Alternately emptying or zeroing your hosts files will have the same effect.

Wrappers first check the hosts.allow file for a rule match. If a match is found, then tcpd stops, depending on the rule, and access is either granted or denied. If no match is found in the hosts.allow file, then tcpd reads the hosts.deny file until a match is found. If a match is found, access is denied, otherwise access is granted.

I have mentioned both the hosts.allow and deny, but due to the flexibility of the rules, you can use just one file, generally hosts.allow, for all of the wrappers rules.

Wrappers will consult the rules in hosts.allow and hosts.deny to determine access. The basic format of the rules are:

```
daemon, daemon, ...  : client, client, ... : option
```

Where:

daemon	The services to be monitored, like telnetd, ftpd, sshd
client	The hosts names, IP address/IP range, or domain names

Option can either be:

allow	Access to the client
deny	Access to the client
except	Will match anything in the first list unless it matches the second list. For example, allow all from domainA except hostX.domainA and hostY.domanA.

Where there are multiple daemons or clients on a line, separate each one with a comma. The ALL keyword can be used to mean ALL daemons or ALL clients.

The LOCAL keyword means to match any host that does not have a dot (".") in it; this means any local host not associated with a domain.

It is good practice and a good habit to append the allow or deny option at the end of each rule where the rule syntax allows (as this provides a clear picture on your access rules, especially when you have multiple allow and deny rules in your hosts.allow file).

There are a few other options as well that I will demonstrate later. For now, let's put some access controls together.

> Changes to the hosts.allow and hosts.deny are dynamic. The changes take effect once the file is saved.

A good starting point is to allow only the clients you want access to the host using the allowed daemons and then deny everything to everybody else.

So, the hosts.deny could deny all daemons to all clients using the following:

```
ALL:ALL
```

The examples in the rest of this section are related to the hosts.allow file only. To allow all daemons access only from local hosts (that is, hosts not associated with a domain name) I could use:

```
ALL: LOCAL : allow
```

On a personal note, I prefer not to use the LOCAL pattern matching on any hosts, as all the hosts in the network should belong to your or some domain. If they are not, then they should be. However, there are occasions with small networks where this may not be the case, and LOCAL allows these hosts access.

Let's now assume that we only wish to allow the hosts that belong to the domain mydomain.com using either telnet or ssh. The following hosts.allow entry accomplishes this:

```
telnetd,sshd: .mydomain.com :allow
```

Notice in this example the dot (".") before mydomain.com. This is a wild card and means all hosts ending with mydomain.com. We also specify at the end of the rule that this is an allow rule. Though not strictly required, as mentioned previously, it is good practice to do so.

Now, further assume that we wish to allow access for ssh and telnet using the following IP addresses: 192.168.4.10 and all IP address that start with 192.168.6. Again, notice the use of the dot after the partial IP address; this equates to 192.168.6.*., or more precisely, all IP addresses that start with 192.168.6. Another way of looking at the IP range 192.168.6. is that it equates to 192.168.6/24 or all IP addresses between 192.168.6.1 thru to 192.168.6.254

Additionally, we also want to allow the following domains access using telnet and ssh. Those domains are mydomain.com and mydomain2.com. The following accomplishes this:

```
telnetd,sshd: .mydomain.com, .mydomain2.com :allow
telnetd,sshd:192.168.4.10 , 192.168.6. : allow
```

Now, assume we wish to allow ftp access from all hosts in the domain mydomain.com, except the following two hosts in mydomain.com: uktrip1 and uktrip2. Using the except option, we can provide two lists where the hosts on the left hand side of the word "except" are granted access, but the hosts contained in the right hand list after the word "except" are denied, by using the allow rule.

```
telnetd,sshd: .mydomain.com :allow
telnetd,sshd:192.168.4.10 , 192.168.6. : allow
ftpd: .mydomain.com except uktrip1.mydomain.com, uktrip2.mydomain.com : allow
```

Let's now look at a deny rule. To refuse telnet to the following IP's, 192.168.8. and 192.168.9., but allow telnet access from 192.168.6., I could use:

```
telnetd : 192.168.8., 192.168.9. : deny
telnetd : 192.168.6. : allow
```

The previous example could also be written using the except option:

```
telnetd: 192.168.6. except 192.168.8. , 192.168.9. : allow
```

Wrappers log the messages to the /var/adm/messages file. A typical refused connection with telnet from a client called tardis in the messages file could be:

```
Oct 23 15:50:55 rs6000 auth|security|warning telnetd[270546]: refused connect from
 tardis
```

A typical ssh failed connection from a client called tardis could be:

```
Oct 23 15:53:36 rs6000 auth|security:info sshd[262252]: refused connect from tardis
```

If you decided to have PARANOID on, wrappers informs you of any IP to host discrepancies it cannot resolve:

```
error ftpd[2605110]: warning: /etc/hosts.allow, line 2: host name/address mismatch:
192.168.7.12 != uktrn004.mydomain.com
```

Sometimes it is a good idea to see which hosts do not have the correct DNS entry, so they can be corrected by those responsible for your DNS. This is especially true in a company internal network. Instead of denying mismatched hosts/IPs, allow only domain users in (one assumes this is in a secure company network). In the following example, all domain users who belong to mydomain.com are allowed access, note the use of ALL for all daemons:

```
ALL:PARANOID, mydomain:allow
```

Using the diag tools tcpdmatch and tcpdchk

The utility tcpdchk checks for syntax errors in your hosts.allow and hosts.deny files. It also tries to resolve IP addresses in the files to make sure they exist, and checks the daemons present in the access files against the /etc/inetd.conf file. The basic format of tcpdchk is:

```
tcpdchk –a | -v
```

Where:

-a	Displays the line that has an error or an ambiguous entry.
-v	Checks and displays all rules.

For example, using the all rule check:

```
# tcpdchk –v
Using network configuration file: /etc/inetd.conf

>>> Rule /etc/hosts.allow line 4:
daemons:  telnetd sshd
warning: /etc/hosts.allow, line 4: sshd: no such process name in /etc/inetd.conf
clients:  192.168.4.10 192.168.5.
access:   granted

>>> Rule /etc/hosts.allow line 5:
daemons:  ftpd
clients:  .mydomain.com EXCEPT uktrip1.mydomain.com uktrip2.mydomain.com
access:   granted

>>> Rule /etc/hosts.deny line 1:
daemons:  ALL
clients:  ALL
option:   banners /etc/banners/deny
option:   DENY
access:   denied
```

Tcpdchk reports that in the first rule entry output the sshd service is not ran from /etc/inetd.conf, which is OK. This version of sshd was built with wrapper support and lives in /usr/sbin. The second rule entry output reports that there are no errors and that the domains are resolvable via the DNS. The third rule entry output from the host.deny file reports no errors either.

The utility tcpdmatch is a good tool for initially testing access rules before putting them in place. Based on the rules provided in the hosts.allow and hosts.deny files, wrappers predicts if access is granted or denied. You simply provide it with a daemon name and a IP or host or domain name that you wish to test.

The basic format for tcpdmatch is:

```
tcpdmatch <daemon> <host>
```

Where <daemon> is the actual daemon, for example ftpd, and <host> is the IP or hostname

Assume we only have the following access rule in our hosts.allow:

```
telnetd,sshd:192.168.4. : allow
```

Tcpdmatch is now ran, parsing it the telnetd daemon name with the IP: 192.168.4.12. The following is output:

```
# tcpdmatch telnetd 192.168.4.12
tcpdmatch telnetd 192.168.4.12
client:   address  192.168.4.12
server:   process  telnetd
matched:  /etc/hosts.allow line 1
option:   allow
access:   granted
```

In this output, the rule states that the IP 192.168.4.12 (with telnetd) is matched against the IP range 192.168.4., thus access is granted.

Now, if I parse it the ftpd daemon with an IP: 192.16.4.12, this is the result:

```
# tcpdmatch ftpd 192.168.4.12
client:   address  192.168.4.12
server:   process  ftpd
matched:  /etc/hosts.deny line 1
option:   DENY
access:   denied
```

In the above output, tcpdmatch correctly states that access is denied, as there is no access rule for ftpd with clients that have an IP ending with 192.168.4.

Now, if I parse it the sshd daemon/service with an IP : 192.168.6.32, the result is:

```
# tcpdmatch sshd 192.168.6.32
warning: sshd: no such process name in /etc/inetd.conf
client:   address  192.168.6.32
server:   process  sshd
matched:  /etc/hosts.deny line 1
option:   DENY
access:   denied
```

> If wrappers is not behaving as you think it should, by denying or allowing the wrong hosts, remember wrappers exit on the first rule match; the ordering of the rules is important. Also use tcpdchk to test the rules against the clients hosts, IP addresses, where you are having issues.

In this output, tcpdmatch correctly states that access is denied, as the IP: 192.168.6.32 (with sshd), is not part of the IP: 192.168.4. range.

Using banners

Wrappers allow the use of banners, as compiled in this demonstration. This allows you to put a message up to the incoming client when a connection is made, or when a connection is denied. I suggest only using banners for denied access. You have the /etc/security/login.cfg herald stanza to put an authorization warning message at the login screen, or you could use /etc/motd once a client is connected.

To create a banner for denied access, first create the /etc/banners/deny directory.

Now, within that directory, create a file for each service daemon. Each file should contain a warning message. Where the filename is the same as the daemon, you are going to use banner messages. So for the telnetd daemon, a file called telnetd would be created. And for the ftp daemon, the file ftpd would be created. By the very nature of how ssh works, and where the ssh client honors true ssh protocol, a banner message will probably not be displayed on a client's ssh connection. The message displayed to the unwanted caller before the connection is broken will be down to your own security policy. However, one could have a message as shown below:

```
Your connection has been refused.
Access denied and the event is logged
```

To confirm, the banner files in this demonstration are located in:

```
# pwd
/etc/banners/deny
# ls
ftpd      telnetd

# cat telnetd
Your connection has been refused.
Access denied and the event is logged
```

Next, inform wrappers of this new configuration addition. Edit the hosts.deny file so you now have the following entry:

```
All:ALL: banners /etc/banners/deny :DENY
```

Now when a telnet or ftp client connection is made and is refused access, and the banner(s) file is present with the daemon name, the following will be displayed, when a denied client tries to connect:

```
$ tn rs6000
Your connection has been refused.
Access denied and the event is logged
```

Twist and spawn

The examples in this section are related to the hosts.allow file only.

In the previous section we looked at banners. However, one can also use the twist option to send out messages to a denied client. The basic format of the twist option is:

```
twist 'shell command'
```

Twist, by its very action, denies access. Twist actually replaces the current service with a specified action. In the following example, the echo command is used to put a denial messages up to a denied client. The example sends a message to the client IP 192.168.9.14, if the client tries to connect using telnet or ssh.

```
sshd,telnetd:192.168.9.14: twist /bin/echo "Your connection has been refused\
\nAccess denied and the event is logged"
```

Using the spawn option is similar to twist except no replies are sent back to the client. Spawn allows a shell command to be spawned. Typically, this would be used to either email or log a message to a log file about a certain host(s) being denied or allowed. The basic format of the spawn option is:

```
spawn '(shell command)'
```

The spawn option is frequently used to contain more than one command. The use of the brackets ensures the commands are grouped and executed in a subshell.

Looking at an example, suppose a connection is made from a test server into production, one would certainly want to be informed about this event, even if the company security policy allows it.

In the following example, all ssh and telnet connections coming from the client that has the IP address 192.168.9.24 are allowed access. Once the client connects, the logger command is executed. Notice in the rule entry that the colon (':') is escaped with a backslash; this preserves the value of the next character. This is required because we do not want wrappers to think this is a separator, which could be part of the access rule. In fact, it is part of the message sent with logger:

```
telnetd,sshd:192.168.9.24 : spawn (/usr/bin/logger 'DEV BOX!!Warning!!\:
 %a has connected') :allow
```

Notice also in the previous entry the use of special shell expansion variable %a, these are prefixed with a '%'. These can be used to include client information, if available, in your messages. Common shell variables are:

- %a - The clients IP address
- %h - The clients hostname
- %c - The clients username

Upon a successful connection, logger via syslog sends the message to /var/adm/messages:

```
Oct 24 15:47:22 rs6000 user:notice root: DEV BOX!!Warning!!: 192.168.9.24 has connected
Oct 24 15:47:22 rs6000 auth|security telnetd[245806]: connect from uk01dev002
```

You can also change the previous rule around to deny that IP address and use logger to send a message via syslog, for example:

```
telnetd,sshd:192.168.9.24 : spawn (/usr/bin/logger '* DEV BOX *!!Warning!!\:
 %a has tried to connected') :deny
```

Upon a denied attempt, logger via syslog sends the message to /var/adm/messages:

```
Oct 24 20:02:39 rs6000 user:notice root: * DEV BOX *!!Warning!!: 192.168.9.24
 Tried to connect
Oct 24 20:02:39 rs6000 auth|security|warning telnetd[237744]: refused connect from
 uk01dev002
```

You can also email user(s) using spawn. In the following example, an email is sent to the user secport upon an allowed connection from the client with the IP: 192.168.9.24, which is allowed access using telnet or ssh:

```
telnetd,sshd:192.168.9.24 : spawn (/usr/bin/echo 'DEV BOX!Warning %a has connected'
 |/usr/bin/mail secport) :allow
```

Conclusion

Using TCP Wrappers, or wrappers, is an easy way to protect and secure your machines based on daemon use and where the client is coming from. Assuming you have the same hardware, rolling it out is simple, just scp all the binaries and your hosts files out to all remote hosts. The only manual configuration required will be the editing of /etc/inetd.conf, but the sed utility can take of that within a script.

Resources

Get products and technologies

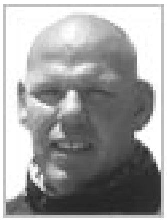
- Download TCP Wrappers IPv4 source
- Download TCP Wrappers binary version 7.6.1
- Download gcc from the IBM AIX toolbox
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

Discuss

- Follow developerWorks on Twitter.
- Participate in developerWorks blogs and get involved in the developerWorks community.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX® forums:
  - AIX Forum
  - AIX Forum for developers
  - Cluster Systems Management
  - Performance Tools Forum
  - Virtualization Forum
  - More AIX and UNIX Forums

About the author

David Tansley



David Tansley is a freelance writer. He has 15 years of experience as a UNIX administrator, using AIX the last eight years. He enjoys playing badminton, then relaxing watching Formula 1, but nothing beats riding and touring on his GSA motorbike with his wife.