NAME

snmpd.examples - example configuration for the Net-SNMP agent

DESCRIPTION

The *snmpd.conf(5)* man page defines the syntax and behaviour of the various configuration directives that can be used to control the operation of the Net-SNMP agent, and the management information it provides.

This companion man page illustrates these directives, showing some practical examples of how they might be used.

AGENT BEHAVIOUR

Listening addresses

The default agent behaviour (listing on the standard SNMP UDP port on all interfaces) is equivalent to the directive:

```
agentaddress udp:161
```

or simply

```
agentaddress 161
```

The agent can be configured to *only* accept requests sent to the local loopback interface (again listening on the SNMP UDP port), using:

```
agentaddress localhost:161     # (udp implicit)
```

or

```
agentaddress 127.0.0.1     # (udp and standard port implicit)
```

It can be configured to accept both UDP and TCP requests (over both IPv4 and IPv6), using:

```
agentaddress udp:161,tcp:161,udp6:161,tcp6:161
```

Other combinations are also valid.

Run-time privileges

The agent can be configured to relinquish any privileged access once it has opened the initial listening ports. Given a suitable "snmp" group (defined in */etc/group*), this could be done using the directives:

```
agentuser  nobody
agentgroup snmp
```

A similar effect could be achieved using numeric UID and/or GID values:

```
agentuser  #10
agentgroup #10
```

SNMPv3 Configuration

Rather than being generated pseudo-randomly, the engine ID for the agent could be calculated based on the MAC address of the second network interface (*eth1*), using the directives:

```
engineIDType 3 engineIDNic  eth1
```

or it could be calculated from the (first) IP address, using:

```
engineIDType 1
```

or it could be specified explicitly, using:

```
engineID "XXX - WHAT FORMAT"
```

ACCESS CONTROL

SNMPv3 Users

The following directives will create three users, all using exactly the same authentication and encryption settings:

```
createUser me     MD5 "single pass phrase"
createUser myself MD5 "single pass phrase" DES
createUser andI   MD5 "single pass phrase" DES "single pass phrase"
```

Note that this defines three *distinct* users, who could be granted different levels of access. Changing the passphrase for any one of these would not affect the other two.

Separate pass phrases can be specified for authentication and encryption:

```
createUser onering SHA "to rule them all" AES "to bind them"
```

Remember that these *createUser* directives should be defined in the /var/lib/snmp/snmpd.conf file, rather than the usual location.

Traditional Access Control

The SNMPv3 users defined above can be granted access to the full MIB tree using the directives:

```
rouser me
rwuser onering
```

Or selective access to individual subtrees using:

```
rouser myself   .1.3.6.1.2
rwuser andI     system
```

Note that a combination repeating the same user, such as:

```
rouser onering
rwuser onering
```

should **not** be used. This would configure the user *onering* with read-only access (and ignore the *rwuser* entry altogether). The same holds for the community-based directives.

The directives:

```
rocommunity public
rwcommunity private
```

would define the commonly-expected read and write community strings for SNMPv1 and SNMPv2c requests. This behaviour is **not** configured by default, and would need to be set up explicitly.

Note: It would also be a very good idea to change *private* to something a little less predictable!

A slightly less vulnerable configuration might restrict what information could be retrieved:

```
rocommunity public   default system
```

or the management systems that settings could be manipulated from:

```
rwcommunity private  10.10.10.0/24
```

or a combination of the two.

VACM Configuration

This last pair of settings are equivalent to the full VACM definitions:

```
#       sec.name  source        community
com2sec   public    default       public
com2sec   mynet     10.10.10.0/24 private
com2sec6  mynet     fec0::/64     private

#               sec.model  sec.name
group  worldGroup  v1         public
group  worldGroup  v2c        public
group  myGroup     v1         mynet
group  myGroup     v2c        mynet

#           incl/excl   subtree     [mask]
view   all     included   .1
view   sysView included   system

#           context model level   prefix  read    write  notify (unused)
access  worldGroup  ""  any  noauth  exact   system  none   none
access  myGroup     ""  any  noauth  exact   all     all    none
```

There are several points to note in this example:

The *group* directives must be repeated for both SNMPv1 and SNMPv2c requests.

The *com2sec* security name is distinct from the community string that is mapped to it. They can be the same ("public") or different ("mynet"/"private") - but what appears in the *group* directive is the security name, regardless of the original community string.

Both of the *view* directives are defining simple OID subtrees, so neither of these require an explicit mask. The same holds for the "combined subtree2 view defined below. In fact, a mask field is only needed when defining row slices across a table (or similar views), and can almost always be omitted.

In general, it is advisible not to mix traditional and VACM-based access configuration settings, as these can sometimes interfere with each other in unexpected ways. Choose a particular style of access configuration, and stick to it.

Typed-View Configuration

A similar configuration could also be configured as follows:

```
view   sys2View included    system
view   sys2View included    .1.3.6.1.2.1.25.1

authcommunity read      public  default     -v sys2View
authcommunity read,write private 10.10.10.0/8
```

This mechanism allows multi-subtree (or other non-simple) views to be used with the one-line *rocommunity* style of configuration.

It would also support configuring "write-only" access, should this be required.

SYSTEM INFORMATION

System Group

The full contents of the 'system' group (with the exception of `sysUpTime`) can be explicitly configured using:

```
# Override 'uname -a' and hardcoded system OID - inherently read-only values
sysDescr     Universal Turing Machine mk I
sysObjectID  .1.3.6.1.4.1.8072.3.2.1066

# Override default values from 'configure' - makes these objects read-only
sysContact   Alan.Turing@pre-cs.man.ac.uk
sysName      tortoise.turing.com
sysLocation  An idea in the mind of AT

# Standard end-host behaviour
sysServices  72
```

Host Resources Group

The list of devices probed for potential inclusion in the `hrDiskStorageTable` (and `hrDeviceTable`) can be amended using any of the following directives:

```
ignoredisk /dev/rdsk/c0t2d0
```

which prevents the device */dev/rdsk/c0t2d0* from being scanned,

```
ignoredisk /dev/rdsk/c0t[!6]d0
ignoredisk /dev/rdsk/c0t[0-57-9a-f]d0
```

either of which prevents all devices */dev/rdsk/c0t*X*d0* (except .../*c0t6d0*) from being scanned,

```
ignoredisk /dev/rdsk/c1*
```

which prevents all devices whose device names start with */dev/rdsk/c1* from being scanned, or

```
ignoredisk /dev/rdsk/c?t0d0
```

which prevents all devices */dev/rdsk/c*X*t0d0* (where 'X' is any single character) from being scanned.

Process Monitoring

The list of services running on a system can be monitored (and provision made for correcting any problems), using:

```
# At least one web server process must be running at all times
proc    httpd
procfix httpd  /etc/rc.d/init.d/httpd restart

# There should never be more than 10 mail processes running
#    (more implies a probable mail storm, so shut down the mail system)
proc    sendmail   10
procfix sendmail  /etc/rc.d/init.d/sendmail stop
```

```
# There should be a single network management agent running
#  ("There can be only one")
proc  snmpd  1  1
```

Also see the "DisMan Event MIB" section later on.

Disk Usage Monitoring

The state of disk storage can be monitored using:

```
includeAllDisks 10%
disk /var 20%
disk /usr  3%
#  Keep 100 MB free for crash dumps
disk /mnt/crash  100000
```

System Load Monitoring

A simple check for an overloaded system might be:

```
load 10
```

A more refined check (to allow brief periods of heavy use, but recognise sustained medium-heavy load) might be:

```
load 30 10 5
```

Log File Monitoring

TODO

```
file FILE [MAXSIZE]
logmatch NAME PATH CYCLETIME REGEX
```

ACTIVE MONITORING

Notification Handling

Configuring the agent to report invalid access attempts might be done by:

```
authtrapenable 1
trapcommunity  public
trap2sink      localhost
```

Alternatively, the second and third directives could be combined (and an acknowledgement requested) using:

```
informsink     localhost  public
```

A configuration with repeated sink destinations, such as:

```
trapsink       localhost
trap2sink      localhost
informsink     localhost
```

should **NOT** be used, as this will cause multiple copies of each trap to be sent to the same trap receiver.

TODO - discuss SNMPv3 traps

```
trapsess  snmpv3 options  localhost:162
```

TODO - mention trapd access configuration

DisMan Event MIB

The simplest configuration for active self-monitoring of the agent, by the agent, for the agent, is probably:

```
# Set up the credentials to retrieve monitored values
createUser    _internal MD5 "the first sign of madness"
iquerySecName _internal
rouser        _internal

# Active the standard monitoring entries
defaultMonitors        yes
linkUpDownNotifications yes

# If there's a problem, then tell someone!
trap2sink localhost
```

The first block sets up a suitable user for retrieving the information to by monitored, while the following pair of directives activates various built-in monitoring entries.

Note that the DisMan directives are not themselves sufficient to actively report problems - there also needs to be a suitable destination configured to actually send the resulting notifications to.

A more detailed monitor example is given by:

```
monitor −u me −o hrSWRunName "high process memory" hrSWRunPerfMem > 10000
```

This defines an explicit boolean monitor entry, looking for any process using more than 10MB of active memory. Such processes will be reported using the (standard) DisMan trap`mteTriggerFired`, b ut adding an extra (wildcarded) varbind `hrSWRunName`.

This entry also specifies an explicit user (*me*, as defined earlier) for retrieving the monitored values, and building the trap.

Objects that could potentially fluctuate around the specified level are better monitored using a threshold monitor entry:

```
monitor −D −r 10 "network traffic" ifInOctets 1000000 5000000
```

This will send a `mteTriggerRising` trap whenever the incoming traffic rises above (roughly) 500 kB/s on any network interface, and a corresponding `mteTriggerFalling` trap when it falls below 100 kB/s again.

Note that this monitors the deltas between successive samples (*−D*) rather than the actual sample values themselves. The same effect could be obtained using:

```
monitor −r 10 "network traffic" ifInOctets − − 1000000 5000000
```

The *linkUpDownNotifications* directive above is broadly equivalent to:

```
notificationEvent  linkUpTrap   linkUp   ifIndex ifAdminStatus ifOperStatus
notificationEvent  linkDownTrap  linkDown ifIndex ifAdminStatus ifOperStatus

monitor  −r 60 −e linkUpTrap   "Generate linkUp"   ifOperStatus != 2
monitor  −r 60 −e linkDownTrap "Generate linkDown" ifOperStatus == 2
```

This defines the traps to be sent (using *notificationEvent*), and explicitly references the relevant notification in the corresponding monitor entry (rather than using the default DisMan traps).

The *defaultMonitors* directive above is equivalent to a series of (boolean) monitor entries:

```
monitor  −o prNames      −o prErrMessage  "procTable" prErrorFlag   != 0
monitor  −o memErrorName −o memSwapErrorMsg "memory"  memSwapError  != 0
monitor  −o extNames     −o extOutput     "extTable"  extResult     != 0
monitor  −o dskPath      −o dskErrorMsg   "dskTable"  dskErrorFlag  != 0
monitor  −o laNames      −o laErrMessage  "laTable"   laErrorFlag   != 0
monitor  −o fileName     −o fileErrorMsg  "fileTable" fileErrorFlag != 0
```

and will send a trap whenever any of these entries indicate a problem.

An alternative approach would be to automatically invoke the corresponding "fix" action:

```
setEvent   prFixIt  prErrFix = 1
monitor −e prFixIt "procTable" prErrorFlag   != 0
```

(and similarly for any of the other *defaultMonitor* entries).

DisMan Schedule MIB

The agent could be configured to reload its configuration once an hour, using:

```
repeat 3600 versionUpdateConfig.0 = 1
```

Alternatively this could be configured to be run at specific times of day (perhaps following rotation of the logs):

```
cron 10 0 * * * versionUpdateConfig.0 = 1
```

The one-shot style of scheduling is rather less common, but the secret SNMP virus could be activated on the next occurance of Friday 13th using:

```
at   13 13 13 * 5 snmpVirus.0 = 1
```

EXTENDING AGENT FUNCTIONALITY

Arbitrary Extension Commands

Old Style

exec [MIBOID] NAME PROG ARGS"
sh [MIBOID] NAME PROG ARGS"
execfix NAME PROG ARGS"

New Style

extend [MIBOID] NAME PROG ARGS"
extendfix [MIBOID] NAME PROG ARGS"

MIB-Specific Extension Commands

One-Shot

"pass [−p priority] MIBOID PROG"

Persistent

"pass_persist [−p priority] MIBOID PROG"

Embedded Perl Support

If embedded perl support is enabled in the agent, the default initialisation is equivalent to the directives:

disablePerl false
perlInitFile /usr/share/snmp/snmp_perl.pl

The main mechanism for defining embedded perl scripts is the *perl* directive. A very simple (if somewhat pointless) MIB handler could be registered using:

perl use Data::Dumper;
perl sub myroutine { print "got called: ",Dumper(@_),"\n"; }
perl $agent−>register('mylink', '.1.3.6.1.8765', \&myroutine);

This relies on the *$agent* object, defined in the example `snmp_perl.pl` file.

A more realistic MIB handler might be:

XXX - WHAT ???

Alternatively, this code could be stored in an external file, and loaded using:

perl 'do /usr/share/snmp/perl_example.pl';

Dynamically Loadable Modules

TODO

dlmod NAME PATH"

Proxy Support

A configuration for acting as a simple proxy for two other SNMP agents (running on remote systems) might be:

com2sec −Cn rem1context rem1user default remotehost1
com2sec −Cn rem2context rem2user default remotehost2

proxy −Cn rem1context −v 1 -c public remotehost1 .1.3
proxy −Cn rem2context −v 1 -c public remotehost2 .1.3

(plus suitable access control entries).

The same *proxy* directives would also work with (incoming) SNMPv3 requests, which can specify a context directly. It would probably be more sensible to use contexts of *remotehost1* and *remotehost2* - the names above were chosen to indicate how these directives work together.

Note that the administrative settings for the proxied request are specified explicitly, and are independent of the settings from the incoming request.

An alternative use for the *proxy* directive is to pass part of the OID tree to another agent (either on a remote host or listening on a different port on the same system), while handling the rest internally:

proxy −v 1 −c public localhost:6161 .1.3.6.1.4.1.99

This mechanism can be used to link together two separate SNMP agents.

A less usual approach is to map one subtree into a different area of the overall MIB tree (either locally or on

a remote system):

```
# uses SNMPv3 to access the MIB tree .1.3.6.1.2.1.1 on 'remotehost'
# and maps this to the local tree .1.3.6.1.3.10
proxy -v 3 -l noAuthNoPriv -u user remotehost .1.3.6.1.3.10 .1.3.6.1.2.1.1
```

SMUX Sub-Agents

```
smuxsocket 127.0.0.1
smuxpeer .1.3.6.1.2.1.14 ospf_pass
```

AgentX Sub-Agents

The Net-SNMP agent could be configured to operate as an AgentX master agent (listening on a non-standard named socket, and running using the access privileges defined earlier), using:

```
master agentx
agentXSocket /tmp/agentx/master
agentXPerms  0660 0550 nobody snmp
```

A sub-agent wishing to connect to this master agent would need the same *agentXSocket* directive, or the equivalent code:

```
netsnmp_ds_set_string(NETSNMP_DS_APPLICATION_ID, NETSNMP_DS_AGENT_X_SOCKET,
                      "/tmp/agentx/master");
```

A loopback networked AgentX configuration could be set up using:

```
agentXSocket   tcp:localhost:705
agentXTimeout  5
agentXRetries  2
```

on the master side, and:

```
agentXSocket   tcp:localhost:705
agentXTimeout  10
agentXRetries  1
agentXPingInterval 600
```

on the client.

Note that the timeout and retry settings can be asymmetric for the two directions, and the sub-agent can poll the master agent at regular intervals (600s = every 10 minutes), to ensure the connection is still working.

OTHER CONFIGURATION

```
override sysDescr.0 octet_str "my own sysDescr"
injectHandler stash_cache NAME table_iterator
```

FILES

/etc/snmp/snmpd.conf

SEE ALSO

snmpconf(1), snmpd.conf(5), snmp.conf(5), snmp_config(5), snmpd(8), EXAMPLE.conf, netsnmp_config_api(3).