by Frédéric Raynal

About the author:
Frédéric Raynal is preparing a thesis about watermarking of digital images at the INRIA (Institut National de Recherche en Informatique et Automatique). *Content*:

xinetd

Abstract:

xinetd − eXtended InterNET services daemon − provides a good security against intrusion and reduces the risks of *Denial of Services* (*DoS*) attacks. Like the well known couple (`inetd+tcpd`), it enables the configuration of the access rights for a given machine, but it can do much more. In this article we will discover its many features.

What's that, xinetd ?

The classical `inetd` helps controlling network connections to a computer. When a request comes to a port managed by `inetd`, then inetd forwards it to a program called tcpd. Tcpd decides, in accordance with the rules contained in the `hosts.{allow, deny}` files whether or not to grant the request. If the request is allowed, then the the corresponding server process (e.g ftp) can be started. This mechanism is also referred to as `tcp_wrapper`.

xinetd provides access control capabilities similar to the ones provided by `tcp_wrapper`. However, its capabilities extend much further :

- access control for TCP, UDP and RPC services (the latter ones aren't well supported yet).
- access control based on time segments
- full logging, either for connection success or failure
- efficient containment against *Denial of Services* (DoS) attacks (attacks which attempt to freeze a machine by saturating its resources) :
    - limitation on the number of servers of the same type to run at a time
    - limitation on the total number of servers
    - limitation on the size of the log files.
- binding of a service to a specific interface: this allows you, for instance, to make services available to your private network but not to the outside world.
- can be used as a proxy to other systems. Quite useful in combination with ip_masquerading (or Network Address Translation − NAT) in order to reach the internal network.

The main drawback, as already mentioned, concerns poorly supported RPC calls. However, `portmap` can coexist with xinetd to solve this.

The first part of this article explains how xinetd works. We'll spend some time on a service configuration, on some specific options (binding to an interface, redirection) and demonstrate this with a few examples. The second part shows xinetd at work, the logs it generates and finishes with a useful tip.

Compilation & Installation

You can get xinetd from www.xinetd.org. For this article we will use version 2.1.8.9pre10.
Compilation and installation are done in the classical way: the usual commands `./configure; make; make install` do it all :) `configure` supports the usual options. Three specific options are available at compile time:

1. `--with-libwrap` : with this option, xinetd checks the `tcpd` configuration files (`/etc/hosts.{allow, deny}`) and if access is accepted, it then uses its own control routines. For this option to work, tcp_wrapper and its libraries have to be installed on the machine (Author's note: what can be done with the wrapper can also be done with xinetd. Allowing this compatibility leads to multiplying the config files, and makes the administration heavier... in short, I don't recommend it);
2. `--with-loadavg` : this option allows xinetd to handle the max_load configuration option. This allows the deactivation of some services when the machine is overloaded. An options essential to prevent some DoS attacks (check the attribute `max_load` in table 1);
3. `--with-inet6` : if you feel like using IPv6, this option allows to support it. The IPv4 **and** IPv6 connections are managed, but IPv4 addresses are changed into IPv6 format.

Before starting xinetd, you don't have to stop inetd. Nevertheless, not doing so may lead to an unpredictable behavior of both daemons!

Some signals can be used to modify xinetd behavior:

- SIGUSR1 : software re−configuration : the configuration file is re−read and the services parameters are changed accordingly
- SIGUSR2 : hardware re−configuration: as above, but furthermore, the outdated daemons are killed
- SIGTERM : ends xinetd and the daemons it generated

There are a few others (let's mention a mistake in the documentation and the man pages: SIGHUP writes its dump in the file `/var/run/xinetd.dump` and not in `/tmp/xinetd.dump`), but the three mentioned above can be easily managed with a small script containing the start, stop, restart, soft, hard options (the latter two respectively corresponding to SIGUSR1 and SIGUSR2).

Configuration

The `/etc/xinetd.conf` file is the default configuration file for the xinetd daemon (a command line option allows to provide another one). The xinetd configuration is not very complex, but it may be a long work and the syntax is unfortunately quite different from that of its predecessor `inetd`.

Two utilities (`itox` and `xconv.pl`) are provided with xinetd and allow to convert the `/etc/inetd.conf` file into a configuration file for xinetd. Obviously, that's not enough since the rules specified in the wrapper configuration are ignored. The `itox` program, still maintained, is no longer developed. The `xconv.pl` program is a better solution, even if the result has to be modified because of features that xinetd has in addition to `inetd`:

```
>>/usr/local/sbin/xconv.pl < /etc/inetd.conf >
/etc/xinetd.conf
```

The configuration file begins with a default section. The attributes in this section will be used by every service xinetd manages. After that, you will find as many sections as there are services, each of them being able to re−define specific options in relation to the default ones.

The default values section looks like:

```
defaults
{
    attribute operator value(s)
    ...
}
```

Each attribute defined in this section keeps the provided value(s) for all services described thereafter. Thus, the `only_from` attribute, allows to give a list of authorized addresses that should be able to connect to servers:

```
only_from = 192.168.1.0/24 192.168.5.0/24 192.168.10.17
```

Every service declared thereafter will allow access from machines having an address contained in the list. However, these default values can be modified for each service (check the operators, explained a bit further down). Nevertheless, this process is a bit risky. As a matter of fact, to keep things simple and secure, it's much better not to define default values and change them later on within a service. For instance, talking about access rights, the simplest policy consists in denying access to everyone and next allowing access to each service to those who really need it (with tcp_wrapper, this is done from an `hosts.deny` file containing

`ALL:ALL@ALL`, and an `hosts.allow` file only providing authorized services and addresses).

Each section describing a service in the config file looks like:

```
service service_name
{
    attribute operator value(s)
    ...
}
```

Three operators are available: '=', '+=' and '−=='. Most of the attributes only support the '=' operator, used to assign a fix value to an attribute. The '+=' operator adds an item to a list of values, while the '−=' operator removes this item.

The table 1 briefly describes some of these attributes. We'll see how to use them with a few examples. Reading the `xinetd.confman page` provides more information.

Tab. 1 : a few attributes for xinetd

Attribute	Values and description
`flags`	Only the most current values are mentioned here, check the documentation to find new ones: • IDONLY : only accepts connexions from clients having an identification server; • NORETRY : avoids a new process to be forked again in case of failure; • NAMEINARGS : the first argument of the `server_args` attribute is used as `argv[0]` for the `server`. This allows to use `tcpd` by putting it in the `server` attribute, next writing the server name and its arguments such as `server_args`, as you would do with `inetd`.
`log_type`	xinetd uses `syslogd` and the `daemon.info` selector by default. • SYSLOG selector [level] : allows to choose among `daemon`, `auth`, `user` or `local0-7` from `syslogd` ; • FILE [max_size [absolute_max_size]] : the specified file receives information. The two options set the file size limit. When the size is reached, the first one sends a message to syslogd, the second one stops the logging for this service (if it's a common file − or fixed by default − then various services can be concerned).
`log_on_success`	Different information can be logged when a server starts: • PID : the server's PID (if it's an internal xinetd service, the PID has then a value of 0) ; • HOST : the client address ; • USERID : the identity of the remote user, according to RFC1413 defining identification protocol; • EXIT : the process exit status; • DURATION : the session duration.
`log_on_failure`	Here again, xinetd can log a lot of information when a server can't start, either by lack of resources or because of access rules: • HOST, USERID : like above mentioned ;

	• ATTEMPT : logs an access attempt. This an automatic option as soon as another value is provided; • RECORD : logs every information available on the client.
`nice`	Changes the server priority like the `nice` command does.
`no_access`	List of clients not having access to this service.
`only_from`	List of authorized clients. If this attribute has no value, the access to the service is denied.
`port`	The port associated to the service. If it's also defined in the `/etc/services` file, the 2 port numbers must match.
`protocol`	The specified protocol must exist in the `/etc/protocols` file. If no protocol is given, the service's default one is used instead.
`server`	The path to the server.
`server_args`	Arguments to be given to the server.
`socket_type`	stream (TCP), dgram (UDP), raw (IP direct access) or seqpacket ().
`type`	xinetd can manage 3 types of services : 1. RPC : for those defined in the `/etc/rpc` file... but doesn't work very well; 2. INTERNAL : for services directly managed by xinetd (`echo`, `time`, `daytime`, `chargen` and `discard`) ; 3. UNLISTED : for services not defined either in the `/etc/rpc` file, or in the `/etc/services` file ; Let's note it's possible to combine various values, as we'll see with `servers`, `services` and `xadmin` internal services.
`wait`	Defines the service behavior towards threads. Two values are acceptable: • yes : the service is mono−thread, only one connexion of this type can be managed by the service; • no : a new server is started by xinetd for each new service request according to the defined maximum limit (Warning, by default this limit is infinite).
`cps`	Limits the number of incoming connexions. The first argument is this number itself. When the threshold is exceeded, the service is deactivated for a given time, expressed in seconds, provided with the second argument.
`instances`	Defines the maximum number of servers of a same type able to work at the same time.
`max_load`	This gives really the maximum load for a server (for example, 2 or 2.5). Beyond this limit, requests on this server are rejected.
per_source	Either an integer, or UNLIMITED, to restrict the number of connexion from a same origin to a server

The four last attributes shown in table1 allow to control the resources depending on a server. This is efficient to protect from *Denial of Service* (DoS) attacks (freezing a machine by using all of its resources)

This section presented a few xinetd features. The next sections show how to use it and give some rules to make it work properly.

Access Control

As we have seen previously, you can grant (or forbid) access to your box by using IP addresses. However, xinetd allows more features :

- you can do access control by hostname resolution. When doing this, xinetd does a lookup on the hostname(s) specified _for every connection_, and compares the connecting address to the addresses returned for the hostname(s) ;
- you can do access control by .domain.com. When a client connects, xinetd will reverse lookup the connecting address, and see if it is in the specified domain.

To optimize things, obviously IP addresses are obviously better, that way you avoid name lookup(s) on incoming connections to that service. If you must do access control by the hostname, you can significantly speed things up if you run a local (at least a caching) name server. It's even better if you are using domain sockets to perform your address lookup (don't put a `nameserver` entry in `/etc/resolv.conf`).

Service *defaults*

The `defaults` section allows setting values for an number of attributes (check the documentation for the whole list). Some of these attributes (`only_from`, `no_access`, `log_on_success`, `log_on_failure`, ...) hold simultaneously the values allocated in this section and the ones provided in the services.

By default, denying access to a machine, is the first step of a reliable security policy. Next, allowing access will be configured on a per−service basis. We've seen two attributes allowing to control access to a machine, based on IP addresses: `only_from` and `no_access`. Selecting the second one we write:

```
no_access = 0.0.0.0/0
```

which fully blocks services access. However, if you wish to allow everyone to access `echo (ping)` for instance, you then should write in the echo service:

```
only_from = 0.0.0.0/0
```

Here is the logging message you get with this configuration:

```
Sep 17 15:11:12 charly xinetd[26686]: Service=echo−stream:
only_from list and no_access list match equally the address
192.168.1.1
```

Specifically, the access control is done comparing the lists of addresses contained in both attributes. When the client address matches the both lists, the least general one is preferred. In case of equality, like in our example, xinetd is unable to choose and refuses the connexion. To get rid of this ambiguity, you should have written:

```
only_from = 192.0.0.0/8
```

An easier solution is to only control the access with the attribute:

```
only_from =
```

Not giving a value makes every connexion fail :) Then, every service allows access by means of this same attribute.

Important, not to say essential: in case of no access rules at all (i.e. neither `only_from`, nor `no_access`) for a given service (allocated either directly or with the `default`) section, the access to the service is allowed!

Here is an example of `defaults` :

```
defaults
{
  instances       = 15
  log_type        = FILE /var/log/servicelog
  log_on_success  = HOST PID USERID DURATION EXIT
  log_on_failure  = HOST USERID RECORD
  only_from       =
  per_source      = 5

  disabled = shell login exec comsat
  disabled = telnet ftp
  disabled = name uucp tftp
  disabled = finger systat netstat

  #INTERNAL
  disabled = time daytime chargen servers services xadmin

  #RPC
  disabled = rstatd rquotad rusersd sprayd walld
}
```

among internal services, `servers`, `services`, and `xadmin` allow to manage xinetd. More on this later.

Configuring a service

To configure a service, we need ...nothing :) In fact, everything works like it does with defaults values: you just have to precise the attributes and their value(s) to manage the service. This implies either a change in the defaults values or another attribute for this service.

Some attributes must be present according to the type of service (INTERNAL, UNLISTED ou RPC) :

Tab. 2: required attributes

Attribute	Comment
`socket-type`	Every service.
`user`	Only for non INTERNAL services
`server`	Only for non INTERNAL services
`wait`	Every service.
`protocol`	Every RPC service and the ones not contained in `/etc/services`.
`rpc_version`	Every RPC service.
`rpc_number`	Every RPC service, not contained in `/etc/rpc`.
`port`	Every non RPC service, not contained in `/etc/services`.

This example shows how to define services:

```
service ntalk
{
  socket_type   = dgram
```

```
  wait          = yes
  user          = nobody
  server        = /usr/sbin/in.ntalkd
  only_from     = 192.168.1.0/24
}

service ftp
{
  socket_type  = stream
  wait         = no
  user         = root
  server       = /usr/sbin/in.ftpd
  server_args  = -l
  instances    = 4
  access_times = 7:00-12:30 13:30-21:00
  nice         = 10
  only_from    = 192.168.1.0/24
}
```

Let's note that these services are only allowed on the local network (192.168.1.0/24). Concerning FTP, some more restrictions are expected: only four instances are allowed and the service will be available only during certain segments of time.

Port binding: the *bind* attribute

This attribute allows the binding of a service to a specific IP address. This is only useful when a machine has at least two network interfaces, for example a computer beeing part of a local network and connected to Internet through a separate interface.

For instance, a company wishes to install an FTP server for its employees (to access and read internal documentation). This company wants to provide its clients with an FTP access towards its products: `bind` has been made for this company :) The solution is to define two separate FTP services, one for public access, and a second one for internal company access only. However, xinetd must be able to differentiate them: the solution is to use the `id` attribute. It defines a service in a unique way (when not defined within a service, its value defaults to the name of the service).

```
service ftp
{
  id           = ftp-public
  wait         = no
  user         = root
  server       = /usr/sbin/in.ftpd
  server_args  = -l
  instances    = 4
  nice         = 10
  only_from    = 0.0.0.0/0 #allows every client
  bind         = 212.198.253.142 #public IP address for this
server
}

service ftp
{
  id           = ftp-internal
  socket_type  = stream
```

```
  wait         = no
  user         = root
  server       = /usr/sbin/in.ftpd
  server_args  = -l
  only_from    = 192.168.1.0/24 #only for internal use
  bind         = 192.168.1.1  #local IP address for this
server (charly)
}
```

The use of `bind` will allow to call the corresponding daemon, according to the destination of the packets. Thus, with this configuration, a client on the local network must give the local address (or the associated name) to access internal data. In the log file, you can read:

```
00/9/17@16:47:46: START: ftp-public pid=26861
from=212.198.253.142
00/9/17@16:47:46: EXIT: ftp-public status=0 pid=26861
duration=30(sec)
00/9/17@16:48:19: START: ftp-internal pid=26864
from=192.168.1.1
00/9/17@16:48:19: EXIT: ftp-internal status=0 pid=26864
duration=15(sec)
```

The first part comes from the command `ftp 212.198.253.142`, while the second part is about the command from `charly` to itself: `ftp 192.168.1.1`.

Obviously, there's a problem: what happens if a machine doesn't have two static IP addresses? This can happen with `ppp` connections or when using the `dhcp` protocol. It seems it would be much better to bind services to interfaces than to addresses. However, this is not yet supported in xinetd and is a real problem (for instance, writing a C module to access an interface or address depends on the OS, and since xinetd is supported on many OSes...). Using a script solves the problem:

```
#!/bin/sh

PUBLIC_ADDRESS=`/sbin/ifconfig $1 | grep "inet addr" | awk
'{print $2}'| awk -F: '{print $2}'`
sed s/PUBLIC_ADDRESS/"$PUBLIC_ADDRESS"/g /etc/xinetd.base >
/etc/xinetd.conf
```

This script takes the `/etc/xinetd.base` file, containing the desired configuration with PUBLIC_ADDRESS as a replacement for the dynamic address, and changes it in `/etc/xinetd.conf`, modifying the `PUBLIC_ADDRESS` string with the address associated to the interface passed as an argument to the script. Next, the call to this script depends on the type of connection: the simplest is to add the call into the right `ifup-*` file and to restart xinetd.

Service redirection towards an other machine: the *redirect* attribute

xinetd can be used as a transparent proxy, sort of (well, almost ... as we'll see it later) with the redirect attribute. It allows to send a service request towards an other machine to the desired port.

```
service telnet
{
  flags  = REUSE
```

```
  socket_type = stream
  wait  = no
  user  = root
  server = /usr/sbin/in.telnetd
  only_from = 192.168.1.0/24
  redirect = 192.168.1.15 23
}
```

Let's watch what's going on now:

```
>>telnet charly
Trying 192.168.1.1...
Connected to charly.
Escape character is '^]'.


Digital UNIX (sabrina) (ttyp1)

login:
```

At first, the connection seems to be established on `charly`, but the following shows that `sabrina` (an alpha machine, hence "`Digital UNIX`") took over. This mecanism can be both useful and dangerous. When setting it up, logging must be done on both ends of the connection. Furthermore, for this type of service, the use of DMZ and firewall is strongly recommended;−)

Special services

Three services only belong to xinetd. Since these services can't be found in /etc/rpc or /etc/services, they must have the UNLISTED flag (besides the INTERNAL flag informing they are xinetd services)

1. servers: informs about servers in use ;
2. services: informs about available services, their protocol and their port ;
3. xadmin: mixes the functions of the two previous ones.

Obviously, these services make your computer more vulnerable. since they provide important information. Presently, their access is not protected (password protected, for instance). You should use them only at configuration time. Next, in the `defaults` section, you must deny their use:

```
defaults {
  ...
  disabled = servers services xadmin
  ...
}
```

Before activating them, you should take some precautions:

1. The machine running xinetd must be the only one able to connect to these services
2. Limit the number of instances to one
3. Allow access only from the machine running the server.

Let's take the example of the `xadmin` service (the two others can be configured in the same way, apart from the port number ;−) :

```
service xadmin
{
  type  = INTERNAL UNLISTED
  port  = 9100
  protocol = tcp
  socket_type = stream
  wait  = no
  instances = 1
  only_from = 192.168.1.1  #charly
}
```

The `xadmin` service has 5 commands :

1. `help` ...
2. `show run` : like the `servers` service, informs about the presently running servers
3. `show avail` : like the `services` service, informs about the available services (and a bit more)
4. `bye` or `exit` ...

Now, you know they exist: forget them ;−) You can test without these services. Commands such as (`netstat`, `fuser`, `lsof`, ... allow you to know what's going on on your machine, without making it vulnerable as you would when using these services!

Let's play a bit...

Starting with a riddle

Here is a small exercise for the ones who survived ;−) First I will explain configuration used in this exercise and then we will try to find out what happens and why it does not work.

We only need the `finger` service :

```
service finger
{
  flags  = REUSE NAMEINARGS
  server = /usr/sbin/tcpd
  server_args = in.fingerd
  socket_type = stream
  wait  = no
  user  = nobody
  only_from = 192.168.1.1  #charly
}
```

xinetd wasn't compiled with the `−−with−libwrap` option (check the attribute `server`). The `defaults` section is of the same kind of the one previously provided: every access to `charly` is denied wherever the connexion comes from. The `finger` service is not deactivated, nevertheless:

```
pappy@charly >> finger pappy@charly
[charly]
pappy@charly >>

pappy@bosley >>  finger pappy@charly
[charly]
```

```
pappy@bosley >>
```

It seems the request didn't work properly, neither from `charly` (192.168.1.1), an authorized machine, nor from `bosley` (192.168.1.10). Let's have a look at the log files:

```
/var/log/servicelog :
00/9/18@17:15:42: START: finger pid=28857 from=192.168.1.1
00/9/18@17:15:47: EXIT: finger status=0 pid=28857
duration=5(sec)
00/9/18@17:15:55: FAIL: finger address from=192.168.1.10
```

The request from `charly` (the two first lines) works properly according to xinetd: the access is allowed and the request takes 5 seconds. On the other hand, the request from `bosley` is rejected (`FAIL`).
If we look at the configuration of the `finger` service, the server used is not really `in.fingerd`, but the `tcp_wrapper tcpd` service. The wrapper log says:

```
/var/log/services :
Sep 18 17:15:42 charly in.fingerd[28857]: refused connect from
192.168.1.1
```

We see that there's only one line matching our two queries! The one from `bosley` (the second one) was intercepted by xinetd, so it's quite normal not to find it in that log. The selected line really corresponds to the request xinetd allowed, sent from `charly` to `charly` (the first one): time and PID are identical.

Let's summarize what we have:

1. xinetd allowed the request;
2. the `finger` request goes through `tcpd` ;
3. `in.fingerd` rejected this request.

What's going on, then? Since the request is accepted by xinetd, it's sent to the specified server (here `tcpd`). Nevertheless, `tcpd` rejects this connection. Then, we must have a look at `hosts.{allow,deny}`. The `/etc/hosts.deny` file only contains `ALL:ALL@ALL`, what explains why the request has been rejected by the wrapper!

According to the way the `server` and `server_args` service lines have been defined, the wrapper features are still accessible (`banner` − there's a `banner` attribute in xinetd−, `spawn`, `twist`, ...). Remember that the −−with−libwrap compilation option only adds access rights control (with the help of `hosts.{allow,deny}` files), before xinetd process starts. In this example we saw that this configuration allows us to continue using the tcp wrapper features.

This overlapping of features, if it can work, may as well lead to stange behaviors. To use xinetd together with `inetd` and `portmap`, it's much better to manage a service with only one of these "super−daemons".

chroot a service

It's often suggested to restrict the fields of some services, or to create a new environment. The `chroot` command allows to change the root directory for a command (or a script):

```
chroot [options] new_root
```

This is often used to protect services such as bind/DNS or ftp. To duplicate this behavior while benefiting from xinetd features, you have to declare `chroot` as a server. Then, you just have to pass other arguments

via the server_args attribute :)

```
service ftp
{
  id           = ftp
  socket_type  = stream
  wait         = no
  user         = root
  server       = /usr/sbin/chroot
  server_args  = /var/servers/ftp /usr/sbin/in.ftpd -l
}
```

Thus, when a request is sent to this service, the first instruction used is `chroot`. Next, the first argument passed to it is the first one on the `server_args` line, that is the new root. Last, the server is started.

Conclusion

You could now ask yourself which daemon to choose from xinetd or inetd. xinetd offers more features , but it requires a bit more administration, especially until it is included by default in the into distributions (it is now true for most of them). The most secure solution is to use xinetd on machines with public access (like Internet) since it offers a better defense. For machines within a local network `inetd` should be enough.

---

pop3 server

`pop3` seems to be very popular : I recieved mails asking me how to handle it throught `xinetd`. Here is a sample configuration :

```
service pop3
{
        disable = no
        socket_type             = stream
        wait                    = no
        user                    = root
        server                  = /usr/sbin/ipop3d
#       log_on_success          += USERID
#       log_on_failure          += USERID
}
```

Of course, you have to put your own path for the `server` attribute.

the use of `pop3` throught `xinetd` could be painful, depending on the values you use for logging. For instance the use of `USERID` send a request from your `xinetd` to an `identd` server hosted at the pop's client. If no such server is available, a timeout is waited for 30 seconds.

So, when somebody tries to get his mail, he have to wait at least for those 30 seconds if no `identd` server responds. You have to choose between :

1. install an `identd` server on all the clients so your logs are very sharp (take care, one can change the informations provided by `identd`) ;
2. decrease the quality of your logging for that service so that your users could get their mails quickly.

Bad configuration with RH7.0, Mandrake 7.2 and maybe some others ...

bug 24279 send to bugzilla.

Some services configured in `/etc/xinetd.d` are not defined in the file `/etc/services`.

```
[pappy@rootdurum xinetd.d]# grep service *udp
chargen-udp:service chargen-udp
daytime-udp:service daytime-udp
echo-udp:service echo-udp
time-udp:service time
```

I've sublitted a fix ... but the RH's guy doesn't like it ;-(He says it can cause troubles with some other tools such as `chkconfig` and `ntsysv`. If I had to choice between those tools and `xinetd`, I already know what I pick ;-)
Last modified: Wed Feb 28 10:15:27 CET 2001

Talkback form for this article

Every article has its own talkback page. On this page you can submit a comment or look at comments from other readers:

talkback page

Webpages maintained by the LinuxFocus Editor team

LinuxFocus.org
Click here to report a fault or send a comment to LinuxFocus

Translation information:
fr -> -- Frédéric Raynal
fr -> en Georges Tarbouriech

2001-03-01, generated by lfparser version 2.10