|  |  |  |
| --- | --- | --- |
|  | 3/26/2012 |  |
|  |  |
| FreeRADIUS – Best Practice*Best Current Practice (BCP) for Configuration* |
|  |  |  |
|  |  | Alan Buxey |

26/03/2012

FreeRADIUS – Best Practice

Best Current Practice (BCP) for Configuration

In this section we will look at best practices for FreeRADIUS configuration as a Visited or Home organisation. We will not look deeply into local requirements, just the proxying/eduroam aspect.

Note that whilst there are some things that should be done you may find that local requirements, historical configuration methods or version being run may affect your ability to follow certain aspects of this guide. It is written for FreeRADIUS 2.1.12 and should operate with FreeRADIUS 3, but as some parts are specific functions of virtual servers and the built-in language known as ‘unlang’, it is not applicable to v. 1.x.

# Virtual servers (VS)

One of the most compelling reasons to use FreeRADIUS and one of its core assets is the ‘virtual servers’ system. Not to be confused with virtualisation technologies such as VMware or XEN, these virtual servers are distinct entities of the same server on the host which are isolated from each other. Anyone familiar with Apache HTTPd and its virtual server implementation (where [www.example.com](http://www.example.com) and [www.testing.com](http://www.testing.com) could both resolve to the same IP address, but Apache would serve different content depending on the content of the request) will be already familiar with the system. In fact, anyone familiar with the new-world Apache configuration with the ‘available’ and ‘enabled’ directories will know that to enable a virtual server you simply link to it from the available directory into the enabled directory. The same is true for FreeRADIUS virtual servers.

In 2.1.12 there are 3 virtual servers enabled by default after installation (though some distributions may do other things:

* **default** - All normal RADIUS packets will be processed here by default (hence the name)
* **inner-tunnel** - RADIUS packets encapsulated within EAP will be processed here by default
* **control-socket** - A special virtual server that listens on the ‘control’ channel which is bound to a local file socket – it is combined with tools such as ‘radmin’ to change the servers configuration while it is running from the command line (and get details such as status of remote proxies)

There are many other virtual servers available for other tasks such as dealing with accounting in different ways (buffered-sql or decoupled-accounting for example), and even for other functions such as VMPS or DHCP however these are beyond the scope of this document.

If no existing virtual server fits your requirements or you require similar functions but listening on a different IP address, additional virtual servers may be defined which can be optimised for performing particular tasks.

# ‘eduroam’ virtual server for handling your roaming user requests

When an authentication request comes from the National RADIUS Proxy Servers (NRPS), all that a Home site needs to do is authenticate the user. The Home Site should not be looking for any locally significant RADIUS attributes (such as NAS-Identifier or NAS-Port-Type) or returning any special attributes such as VLAN-Id unless a special arrangement is in place with the Visited site. For this reason it is best for the request from the NRPS to be sent to a dedicated virtual server that has been optimised to process authentication requests from remote users.

To have the incoming request sent to a separate virtual server, which we shall define as ‘eduroam’, you need to:

* create the virtual server
* modify the clients.conf entry to point to this new virtual server.

This can be done by creating a copy of the default file in the sites-available directory, naming it ‘eduroam’ and creating a link to it in the sites-enabled directory, as follows:

cd sites-available

cp default eduroam

ln –s eduroam ../sites-enabled/eduroam

Now, edit the new ‘eduroam’ virtual server to give it the ‘eduroam’ name, wrapping the default code with the following:

server eduroam {

 ## Existing default code

}

If you’re unsure about how to do this, take a look at some of the other files in sites-available. The additions to the file ‘names’ the server – this name will show up when debugging, making it easier to trace a packets path. When you restart the FreeRADIUS daemon, the new virtual server is loaded is now ready to be used.

The new ‘eduroam’ virtual server can be stripped down to the minimum required for authorisation, authentication and accounting, plus any relevant logging. As the request will never be proxied away many checks several of the checks you would make for local requests can be skipped, such as queries to LDAP, perl or SQL for assigning VLANs. If you wish to duplicate a virtual-server then use the best match as the starting point; for example, to deal with RADIUS packets copy the default server file, to deal with EAP inner methods copy the inner-tunnel.

To route incoming packets to the new ‘eduroam’ virtual server that has been created, one extra line needs to be added to each client block that represents each of the NRPS:

client roaming2.ja.net {

 secret = secret

 nastype = other

 shortname = NRPS2

}

Becomes

client roaming2.ja.net {

 secret = secret

 nastype = other

 shortname = NRPS2

 virtual\_server = eduroam

}

Any requests from roaming2.ja.net will now be processed by the ‘eduroam’ virtual server.

This needs to be repeated for for roaming1 and roaming0 to ensure that requests incoming from those NRPS are also directed to the ‘eduroam’ virtual server.

If you run the daemon now, you should get no errors. If you do get an error along the lines of “no such virtual server exists”, check the configuration and the virtual server file, as an error in the virtual server file will cause the virtual server to fail to load but the server as a whole may still run.

# Named modules

FreeRADIUS comes with many modules for performing various functions. Most of the modules only perform a single task and need no configuration, however things start to get interesting when you need to use a more complex module in more than one way. For example, there is now another virtual server to process requests from the NRPS – however, modules called by the new virtual server will still have the same configuration options, meaning that there may be unnecessary work being done by the server.. As a more specific example, if the SQL or Perl modules are being used to assign VLANs, then even with the new virtual server there will still be extra, unnecessary processing being done by these modules. To mitigate this, FreeRADIUS allows multiple configurations of the same module. Like virtual servers, modules are given a local name and then called using this new name.

Using SQL to assign VLANS and authenticate users as an example, everything works fine with a single configuration, albeit inefficiently. To create a second configuration for the new virtual server which makes it not do the VLAN assignment, simply copy the current sql.conf file, give the copy a ‘local’ name (such as eduroam-sql.conf or auth-only-sql.conf) and then call that module in the new virtual server instead:

sql {

#sql authentication config here

#sql vlan assignment config here

}

becomes:

sql eduroam-sql {

#sql authentication config here

}

Then update any calls to ‘sql’ in the new ‘eduroam’ virtual server to read ‘eduroam-sql’. There are no specific limitations on the how different the two configurations can be, and could be as extensive as logging to different tables, checking against different checkrad tables, or even connecting to a different database server entirely.

If the file is placed in the modules directory it is activated automatically, otherwise add an $INCLUDE line to radiusd.conf similar to the one that exists for sql.conf.

# Attribute filtering

When a request is sent to a remote RADIUS server, the Access-Accept packet may come with far more than just the Accept attributes. It may also come with VLAN assignment attributes or other “junk” attributes if the remote site isn’t being a nice neighbour (perhaps they don’t have separate virtual servers!). If you use dynamic VLAN assignment for your ‘eduroam’ SSID and you have the same NAS equipment as the remote site – i.e. your access points understand and use the same attributes as the remote site – then this can cause problems, such as users gaining access to parts of the network they shouldn’t.

For this reason, best practice is to filter RADIUS attributes. In FreeRADIUS this can be done in pre\_proxy and post\_proxy stages. Just ensure that you call the attribute filter within each section. By default, its ‘attr\_filter.pre-proxy’ in the pre-proxy section.

# Only proxy valid users

With a default FreeRADIUS installation there is a DEFAULT realm in proxy.conf. It is common for sites to use this in their configuration for proxying requests to the NRPS (i.e. add the 3 National proxies to the DEFAULT realm, scan requests for their home realm, then drop everything else into DEFAULT). This ‘works’ for the end site, however it means that lots of invalid requests reach the National Proxy servers where the requests get rejected. This wastes resources at the national level, and leads to higher load, especially if a client constantly retries the authentication. The correct method is to define a new realm, (call it ‘eduroam’) and then use unlang in the authorize section to send valid requests to this realm. A quick example of this would be :

if( ("%{User-Name}" =~ /\\\\?([^@\\\\]+)@?([-[:alnum:].\_]\*)?$/) && ("%{User-Name}" !~ /@yourrealm.ac.uk$/) ) {

update request {

Realm := "eduroam"

}

update control {

 Proxy-To-Realm := 'eduroam'

 }

}

The ‘FreeRADIUS at Sussex University’ case study covers this in more detail. The DEFAULT realm can then be configured to just drop/reject anything else – this ensures that only valid NAI entries are sent to the NRPS (i.e. entries with valid realm configuration). Coupled with other checks such as looking for trailing whitespace, multiple @, multiple ‘.’ in sequence then only a clean list of authentication requests will be sent to the NRPS. Current misconfigurations are matched by:

/@(.+)?@/i (multiple @ in user-name e.g. user@realm.ac.uk@realm.ac.uk)

/@.\*\\s$/i (trailing whitespace in realm e.g. user@realm.ac.uk )

/\\.\\./ (double dots e.g. user@realm..ac.uk)

/\\.ax\\.uk$/i (invalid realm – a typical typo – e.g. user@realm.ax.uk)

/^@ac\\.uk$/i (invalid realm - no instition e.g. user@ac.uk)

/3gppnetwork\\.org$/i, /myabc\\.com$/i (default realms of some handsets and supplicants)

# Proxy servers

Defining the 3 National RADIUS Proxy Servers (NRPS) in a home\_server pool is part of standard FreeRADIUS 2 configuration, the support server will provide the cut and paste configuration required for best practice – however, if writing your own configuration please note that there are 3 NRPS remote servers and all 3 should be used. To ensure this, the best load-balance method in FreeRADIUS 2 that is compatible with EAP is “client-port-balance” , this is defined in the home\_server\_pool section. For each home\_server definition, “require\_message\_authenticator = yes" should be set (RFC 5080)

# Using policy

Rather than putting all of your rules written in unlang straight into the virtual server server configuration – which can be untidy and difficult to maintain – you can instead write them once in a policy module and then simply call that from the virtual server. If you look at the default policy.conf file you will see a handful of useful nuggets such as ‘permit\_only\_eap’ which only lets EAP-based authentications be processed (ideal for requests coming to your site from the NRPS). There is a default ‘filter\_username’ entry which makes for a good starting point for writing your own rules. Care should be taken when using the default policy modules as some modules, such as the mixed case check are not suitable for eduroam.

 filter\_username {

 # spaces at the start: reject

 if (User-Name =~ /^ /) {

 reject

 }

 # spaces at the end: reject

 if (User-Name =~ / $$/) {

 reject

 }

 }

Once the necessary changes have been made, call ‘filter\_username’ from inside your authorize section (recent FreeRADIUS releases have it there ready to uncomment).

Result of permit\_only\_eap policy check when ‘radtest’ is used:

+- entering group authorize {...}

++- entering policy permit\_only\_eap {...}

+++? if (!EAP-Message)

? Evaluating !(EAP-Message) -> TRUE

+++? if (!EAP-Message) -> TRUE

+++- entering if (!EAP-Message) {...}

++++? if (!"%{outer.request:EAP-Message}")

 expand: %{outer.request:EAP-Message} ->

? Evaluating !("%{outer.request:EAP-Message}") -> TRUE

++++? if (!"%{outer.request:EAP-Message}") -> TRUE

++++- entering if (!"%{outer.request:EAP-Message}") {...}

+++++[reject] returns reject

++++- if (!"%{outer.request:EAP-Message}") returns reject

+++- if (!EAP-Message) returns reject

**++- policy permit\_only\_eap returns reject**

Using Post-Auth-Type Reject

# Reduce inner authentication load

A common issue encountered when trouble-shooting a server is that the server has not been optimised to reduce its load. Often, once a server is in a working state no further work is undertaken to tweak the performance or to verify how efficiently the server is doing its work. If you examine the debugging log, when authenticating an EAP method you will see that the server passes through the inner-tunnel multiple times. This means that every method in authorize is hit many times – when all that really needs to be done is to call the modules once, when the EAP is an ‘Identity’ type – i.e. the server is validating the identity. If you use SQL, LDAP , or any other remote identity store then this means you are hitting the backend authentication server many times rather than just once. As an example, a server using SQL for authentication (cleared up to reduce output):

} # server

server inner-tunnel { # from file /etc/raddb/sites-enabled/inner-tunnel

} # server

Listening on authentication address 127.0.0.1 port 18120 as server inner-tunnel

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 4

rlm\_sql (sql): Released sql socket id: 4

++[sql] returns ok

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 3

rlm\_sql (sql): Released sql socket id: 3

++[sql] returns ok

[peap] TLS\_accept: SSLv3 write server hello A

[peap] TLS\_accept: SSLv3 write server done A

server {

server inner-tunnel {

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 2

rlm\_sql (sql): Released sql socket id: 2

++[sql] returns ok

} # server inner-tunnel

server {

server inner-tunnel {

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 1

rlm\_sql (sql): Released sql socket id: 1

++[sql] returns ok

} # server inner-tunnel

server {

server inner-tunnel {

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 0

rlm\_sql (sql): Released sql socket id: 0

++[sql] returns ok

} # server inner-tunnel

As can be seen, the inner-tunnel ‘sql’ method was hit 3 times (and twice in outer layer). If we now protect the inner-tunnel authentication method so that it is only called when needed with the following configuration:

if ((EAP-Type == 1) || (EAP-Message =~ /^0x02..00061a..$/)) {

 noop

}

else {

 sql

}

Checking the debug log with this change made, and a successful authentication:

} # server

server inner-tunnel { # from file /etc/raddb/sites-enabled/inner-tunnel

} # server

Listening on authentication address 127.0.0.1 port 18120 as server inner-tunnel

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 4

rlm\_sql (sql): Released sql socket id: 4

++[sql] returns ok

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 3

rlm\_sql (sql): Released sql socket id: 3

++[sql] returns ok

[peap] TLS\_accept: SSLv3 write server hello A

[peap] TLS\_accept: SSLv3 write server done A

server {

server inner-tunnel {

} # server inner-tunnel

server {

server inner-tunnel {

[sql] expand: %{User-Name} -> alan

[sql] sql\_set\_user escaped user --> 'alan'

rlm\_sql (sql): Reserving sql socket id: 2

rlm\_sql (sql): Released sql socket id: 2

+++[sql] returns ok

} # server inner-tunnel

There is now just one call to ‘sql’ in inner-tunnel, removing 66% of the overhead. If the server is dealing with many authentications per second this would remove a large amount of load from the authentication server. If the backend server, this would speed up authentications for the end user - if LDAP takes 1 second to query, then any authentications would now be 2 seconds faster, as LDAP is just queried once instead of three times

# Remove unused modules

Another common issue that can affect servers that were previously working fine, and can cause a server to respond very slowly or enter many ‘duplicate’ entries in the log file. This is often a sign that the server isn’t responding to the queries from the NAS in time – particularly accounting packets. If the backend is the source of the delay there little that FreeRADIUS can do (new code in forthcoming versions will deal with overflooding e.g. too many accounting requests by rate limiting the number of packets accepted).

The cause of such slowness is often unnecessary modules which are still active, particularly modules such as ‘radutmp’ and ‘radwtmp’ which are related to simultaneous login and unix authentication. If you do not require these for real operations (eduroam doesn’t have simultaneous usage restrictions) then they should not be enabled – they will happily tick along printing [noop] in your debug log, consuming CPU time and eventually filling up log files with hundreds of megabytes of junk, at which point performance will drop. If you have no use for /etc/passwd authentication, then comment out or remove ‘unix’ module, if you simultaneous usage checking is not needed then comment out the session modules. **Always have a working server before you start to strip out the modules**. It is far easier to add things back one at a time when the authentications no longer work (always perform validations after any change to the configuration) then to try to get a server running from no configuration at all.

# Monitoring

This should not need mentioning as it should be best practice for any service being run, but you should have a full suite of monitoring for your FreeRADIUS server. There are RADIUS check plug-ins for the majority of market-leading monitoring tools that sites use such as NAGIOS, OrionNPM, OpenNMS, Zabbix... It is recommended that as a minimum you run a local authentication and remote authentication check – validating your RADIUS infrastructure is okay for your own users and for visitors. Remote authentication can be tested using <your-realm.ac.uk@eduroam.ac.uk>. Alongside this, monitoring of the server (for example, using Monit or a NAGIOS NRPE instance) will allow you to check the daemon itself and overall system health (especially disk space). FreeRADIUS itself has status information exposed by the control-socket and status virtual servers which will hand out information about the daemon. FreeRADIUS has a MUNIN plugin which can collect information about the FreeRADIUS system and show the details in the MUNIN server graphs, which can be very useful for trend analysis.

