|  |  |  |
| --- | --- | --- |
|  | 2/27/2012 |  |
|  |  |
| FreeRADIUS packet handling*Examining the flow* |
|  |  |  |
|  |  | Alan Buxey |

FreeRADIUS packet handling

Examining the flow

In this section we will look at how a packet goes through the FreeRADIUS server. This fundamental knowledge will help us to understand where and how we configure the server, where we can adjust things or optimise things and where to start looking if things go wrong.

A good starting reference point is the very first configuration after installation - by default this should just work! (There are some reasons why it might not work – for example the server did not get built with EAP support as OpenSSL headers/includes weren’t present – but we assume that the ./configure stage was carefully looked at with all output examined/checked for WARNING messages).

 ./configure *–any\_options\_you\_chose* | grep WARN

The first install configuration will have just a temporary ‘snake oil’ self-signed certificate for EAP but it’s a solid enough start for testing/development. You just need to have a user account to test with. The quickest way of getting one of those is just to do a quick edit of the ‘users’ file, e.g. /usr/local/etc/raddb/users, ensuring that the test account is a valid and unique entry - look at the ‘steve’ entry that comes as an example with the provided users file.

 alan Cleartext-Password := “password”

Using a tool such as ‘eapol\_test’, part of the wpa\_supplicant package, we can fire a test RADIUS request that uses PEAPv0/MSCHAPv2 (Microsoft PEAP) at the server.

For debugging/testing it is recommended that you have multiple terminal windows open to your server so that you can see all that is going on. In window one, ‘radiusd –X’ , in another window use eapol\_test (note that there are plenty of example tests to use with eapol\_test in the FreeRADIUS source directory - freeradius-server-2.1.12/src/tests).

 eapol\_test -c peap-mschapv2.conf -s testing123

This will fire off a test to the localhost (FreeRADIUS by default listens on this for RADIUS packets using port 1812) using the secret ‘testing123’, which is the default secret for localhost client on FreeRADIUS – remember that we haven’t edited any other FreeRADIUS file than the ‘users’ file right now. (Nb. The eapol\_test supplicant file ‘peap-mschapv2.conf’ would have been edited to ensure that the identity and password entries matched the user credentials that you added to the ‘users’ file, e.g.

network={

 ssid="example"

 key\_mgmt=WPA-EAP

 eap=PEAP

 identity="alan"

 anonymous\_identity="alan"

 password="password"

 phase2="auth=MSCHAPV2"

 phase1="peapver=0"

}

In the eapol\_test window, lots of text flies by ending with:

EAPOL: Successfully fetched key (len=32)

PMK from EAPOL - hexdump(len=32): 4e 20 53 15 d2 3b e4 e3 d5 c3 6e 39 56 20 4c f7 3a 94 0a 98 26 e4 6c 80 06 d3 b9 24 8a e2 87 37

EAP: deinitialize previously used EAP method (25, PEAP) at EAP deinit

ENGINE: engine deinit

MPPE keys OK: 1 mismatch: 0

SUCCESS

The FreeRADIUS output window (radius –X) would also have been very busy with the final lines being :

 Sending Access-Accept of id 10 to 127.0.0.1 port 35433

 MS-MPPE-Recv-Key = 0x4e205315d23be4e3d5c36e3956204cf73a940a9826e46c8006d3b9248ae28737

 MS-MPPE-Send-Key = 0xe7f2ba3cf4310fba1bfc021ac1a1c5c4b3d9cba05985a6bc752eef97a75b4085

 EAP-Message = 0x030a0004

 Message-Authenticator = 0x00000000000000000000000000000000

 User-Name = "alan"

Finished request 10.

Going to the next request

Waking up in 4.9 seconds.

Cleaning up request 0 ID 0 with timestamp +16

Cleaning up request 1 ID 1 with timestamp +16

Cleaning up request 2 ID 2 with timestamp +16

Cleaning up request 3 ID 3 with timestamp +16

Cleaning up request 4 ID 4 with timestamp +16

Cleaning up request 5 ID 5 with timestamp +16

Cleaning up request 6 ID 6 with timestamp +16

Cleaning up request 7 ID 7 with timestamp +16

Cleaning up request 8 ID 8 with timestamp +16

Cleaning up request 9 ID 9 with timestamp +16

Cleaning up request 10 ID 10 with timestamp +16

Ready to process requests.

We have a working RADIUS server that can authenticate EAP. So we can use this to analyse how the server operates and understand where we can optimise it or change its behaviour.

The first thing to notice is the copious output of ‘radius –X’ this is the full debug mode but does highlight a few essentials of the server. If we look at the flow of the server we can see the following occurs:

Upon receiving the packet, the server goes into the authorise section of the virtual server, it then goes through each of the modules in the authorise section one by one. Each module can affect the flow or decision made about the packet, if they do nothing then they return ‘noop’, if a module is ALWAYS doing noop and you don’t use it, then you can think about removing it from the configuration. By the end of the authorise section a decision will have been made to either:

* authenticate locally (Auth-Type is set)
* to proxy to a remote server or
* to reject.

In our example, then packet is an EAP request and so we enter the authenticate section with EAP identity and, by default EAP-MD5. The challenge is sent to the client and the return packet arrives…and goes through all the previous steps. The packet is NAK’d with a request for PEAP instead (similar to DHCP NAK and negotiations). The PEAP Access-Challenge is then sent back to the client. Some ping-ponging of Access-Challenge and Access-Request packets then occurs (ALL packets passing through the same authorise section of the default virtual server and then into the authenticate section) until the PEAP session is established:

[peap] Session established. Decoding tunneled attributes

At this point we are now ready to actually deal with the contents inside the EAP tunnel. The server now expects an inner identity and once the packet is received it passes this into the ‘inner-tunnel’ virtual server. Inside the inner-tunnel server the packet enters the authorise section where, once again, it passes through all the modules in the order in which they are listed in the configuration (/etc/raddb/sites-enabled/inner-tunnel), in our example the ‘files’ module is used :

[files] users: Matched entry alan at line 86

The authenticate section is now used – and the eap.conf says that the default for PEAP is MSCHAPv2 so the server issues an MSCHAPv2 challenge to the client. The client responds and this and the reply packet goes into the default virtual server, passing through all the options, then into the inner-tunnel virtual server where it is marked as an ongoing EAP conversation:

[eap] No EAP Start, assuming it's an on-going EAP conversation

The packet then enters the authentication section where the response to the MSCHAPv2 challenge is seen, at which point the server starts the MSCHAP module to create the required hash:

[mschap] Told to do MS-CHAPv2 for alan with NT-Password

This is then sent to the client. The response from the client (which will be the matching answer) is then sent to the server, passes through the default virtual server then through to the inner-tunnel virtual server where it is once again noted to be part of an ongoing conversation and passed to the authentication section.

In the authentication section the EAP module sees that all is okay - ++[eap] returns ok and the post-auth section is passed through. By default nothing is configured for this section but here things such as logging (to a file or SQL) can be configured. PEAP marks the packet as successful and a final challenge packet is sent to the client.

When the client responds the packet passes through the entire system again and then through to the authenticate section of the inner-tunnel where the TLV success/response is finished and finally the Access-Accept packet is seen:

Sending Access-Accept of id 10 to 127.0.0.1 port 54436

For a live working server, all you care about is seeing:

**Sending Access-Accept of id *xxx* to *xxx.xxx.xxx.xxx* port *xxxx***

This means the server has done its job. You can confirm this as a Home site in eduroam, if you can see Access-Accept packets being sent (either by looking for them in debug mode or seeing something like the following in radius.log):

Mon Mar 5 15:35:53 2012 : Auth: Login OK: [user@camford.ac.uk] (from client media-wism port 13 cli 00-00-DE-AD-BE-EF)

If you see this and the client STILL isn’t connecting to the network then there are other problems.If the user isn’t getting online at the remote site then you may need to check things such as – are you sending extra attributes back with the reply? Unless you and the remote site have an agreement on such things as VLANs and RADIUS attributes you may be causing issues – ensure that you use the attribute filter to only send back the bare minimum requires RADIUS attributes.

# Proxying

The difference between the previous authentication (local) and a remote proxied authentication is that in a remote proxy authentication the server doesn’t go into the authentication section. Instead it is informed via the authorise section to proxy the request. In terms of typical configuration this notification would be done via the ‘suffix’ module, if following best practice then it would be done via user-name checking and some unlang to update the request to use a different realm:

[suffix] Looking up realm "camford.ac.uk" for User-Name = "anonymous@camford.ac.uk"

[suffix] Found realm "eduroam"

[suffix] Adding Realm = "eduroam"

[suffix] Proxying request from user anonymous to realm eduroam

[suffix] Preparing to proxy authentication request to realm "eduroam"

The pre-proxy section is then entered after the authorise section (note that all the modules in authorise are still active and whilst EAP will not do anything ([eap] Request is supposed to be proxied to Realm eduroam. Not doing EAP). Other modules such as SQL might not be so benign and may cause the request issues (such as a matching user-name). The pre-proxy section can be configured to do logging, attribute filtering and rewriting or even addition of attributes into the outgoing request (this is the ideal place to add the Operator-Name attribute). The request is then sent off to a remote proxy. When the challenge returns from the remote proxy (which itself would be acting like the Home server to our RADIUS client) the request goes straight into the post-proxy stage of the server. The post-proxy stage can also be used for logging, attribute rewriting and filtering. The Access-Challenge is then sent back to the client.

As a visited site, during the authentication process, the RADIUS server is only passing packets through authorise, pre-proxy and post-proxy. You can imagine this as a thin sandwich layer between the client and the home server. All of the packets sent in the last section locally to establish the PEAP tunnel and to do the authentication still occur – they just get forwarded through a proxy when the client is remote. In fact, as the network is remote and not on the local LAN there may actually be more packets sent since the EAP packets can be large and may be fragmented.

Eventually, the Home server will send the Access-Accept (all being well). The Visited site proxy will see this packet pass once more through the post-proxy section and then a subtle change of behaviour will occur – it will enter the post-auth section (just as a Home server would). At this point such things as setting the VLAN for the client may be configured, more logging can be done or ‘exec’ module used to execute a task.

# Proxy packet handling

When a RADIUS packet is proxied to another server the server needs to track the packet and keep a status view of the remote server. This is to ensure that the remote proxy used is the best one available. In FreeRADIUS 2.o upwards, the server acts in ‘synchronous’ mode – that is it never originates RADIUS packets – i.e. if the remote server does not respond, then FreeRADIUS does not do anything – it only proxies packets or re-transmitted packets that it has received from the NAS (client). If the NAS does not retransmit, then the server may never operate all of the failover mechanisms.

Version 2 of FreeRADIUS added home\_server definitions with load-balancing/failover mechanisms configured in the home\_server\_pool definition. This allows more complex configurations to be easily achieved (you can still use the very basic authhost/accthost single entry for local single servers or internal methods).

When a RADIUS packet is proxied to a remote server, the RADIUS server expects the answer within a time-window known as the ‘response\_window’. By default this is 20 seconds. If a response does not come within this window, then the ‘zombie\_period’ is initiated. If the remote server does not respond to any packets during the zombie period (by default 40 seconds, during which packets can still be sent to the remote server) then it is marked as dead. Note that a server marked as a zombie can still be used but will be used as a low priority..if you have 3 remote servers in the pool then the other 2 will be used in preference with the zombie-marked server only being used if there are no servers marked as alive. Any current request that has been sent to a particular server will still be sent upon retransmit to the zombie server until it is marked as dead, at which point transmission to another server will commence. If there is no way to detect whether a server is dead (covered in the next paragraph) then the server will wait for a period of time (revive\_interval, by default 120s) and then mark the server as alive…if it is NOT alive then the remote server will go back into zombie state again.

The server can detect if the remote server is really dead or not by using status checks. In a multi-tier proxy system, the National Proxies are sending the packets onwards to other sites – if those end sites do not respond then the National Proxies will appear to be ‘dead’ to the local server and it would only take one visitor from one bad home site to send the usual three authentication attempts and your server would think all 3 NRPS are dead. The National RADIUS proxy servers (NRPS) support status checks. By using status-checks, the local server is able to send a special packet up to remote proxy to see if it gets a response. There are 2 basic types – status-server or request. Request will send a real user request (an Access-Request or Accounting-Request depending on the type of server defined) that needs to be dealt with; we strongly recommend the simple ‘status-server’ method. If a remote server has not responded within the response\_window, then the ‘Status-Server’ probe mechanism will kick into action. There is a check\_interval (interval between the status-server probes) and there is a num\_answers\_to\_alive (which is the number of checks in a row that the remote server must respond to). This is shown as follows:



So, once a remote server has been marked as ‘zombie’ the default configuration will take 120s until it can be shown to be alive by status-server method (3 status probes, 30 seconds apart). We would recommend that this be optimized by adjusting the check\_interval to a low settings such as 6 seconds. Do not reduce the check\_interval AND the number of status probes. This would bring the remote server out of zombie state in 18 seconds (just a slightly smaller time than the default response\_window).