freeRADIUS

A High Performance, Open Source, Pluggable, Scalable

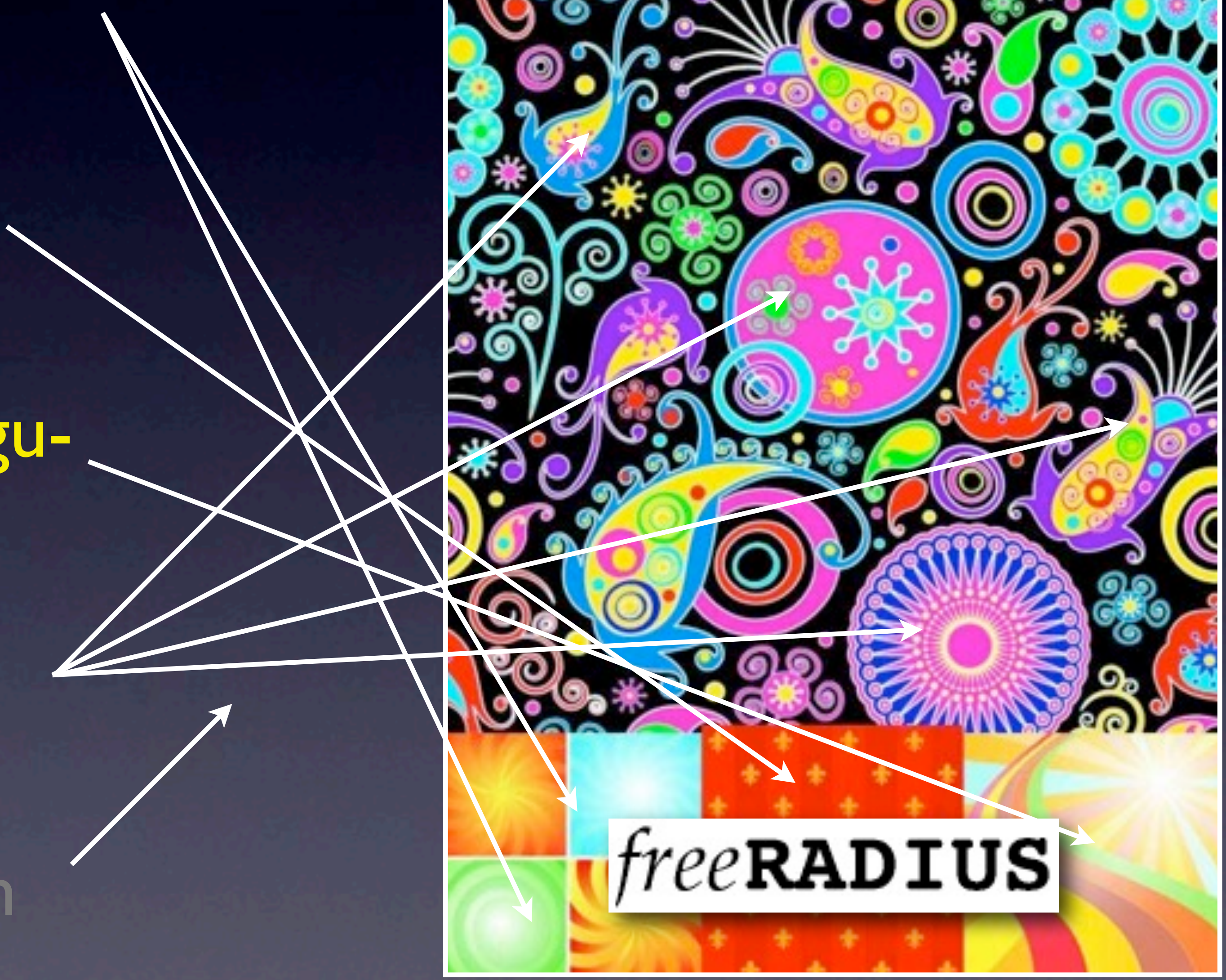
(but somewhat complex)

RADIUS Server

Aurélien Geron, Wifirst, january 7th 2011

Roadmap

- Multiple protocoles : RADIUS, EAP...
- An Open-Source (GPLv2) server
- A powerful configuration system
- Many expansion modules
- Writing your own modules



Source image: http://crshare.com/abstract-backgrounds-vector-clipart/

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
    - Configuration of the RADIUS dictionary
    - Basic configuration of the server
    - Request management policies configuration
    - Modules configuration
    - Roaming configuration

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
  - Configuration of the RADIUS dictionary
  - Basic configuration of the server
  - Request management policies configuration
  - Modules configuration
  - Roaming configuration

The RADIUS dictionary

- Reminder: the name and type of the attributes are *not* actually sent as such in RADIUS packets, only their number and value
- It would be a pain to have to configure freeRADIUS (or any RADIUS client or server) using only attribute numbers
- This is why freeRADIUS (and virtually all RADIUS softwares) use a dictionary that allows you to associate a name and a type to each attribute number, and then use the human-readable name in the rest of the configuration

The RADIUS dictionary

- The file `/etc/freeradius/dictionary` is the entry point to the definition of the RADIUS dictionary used throughout the freeRADIUS configuration

- By default, it just contains one single line (plus some comments) which includes the standard dictionary:

```
$INCLUDE /usr/share/freeradius/dictionary
```

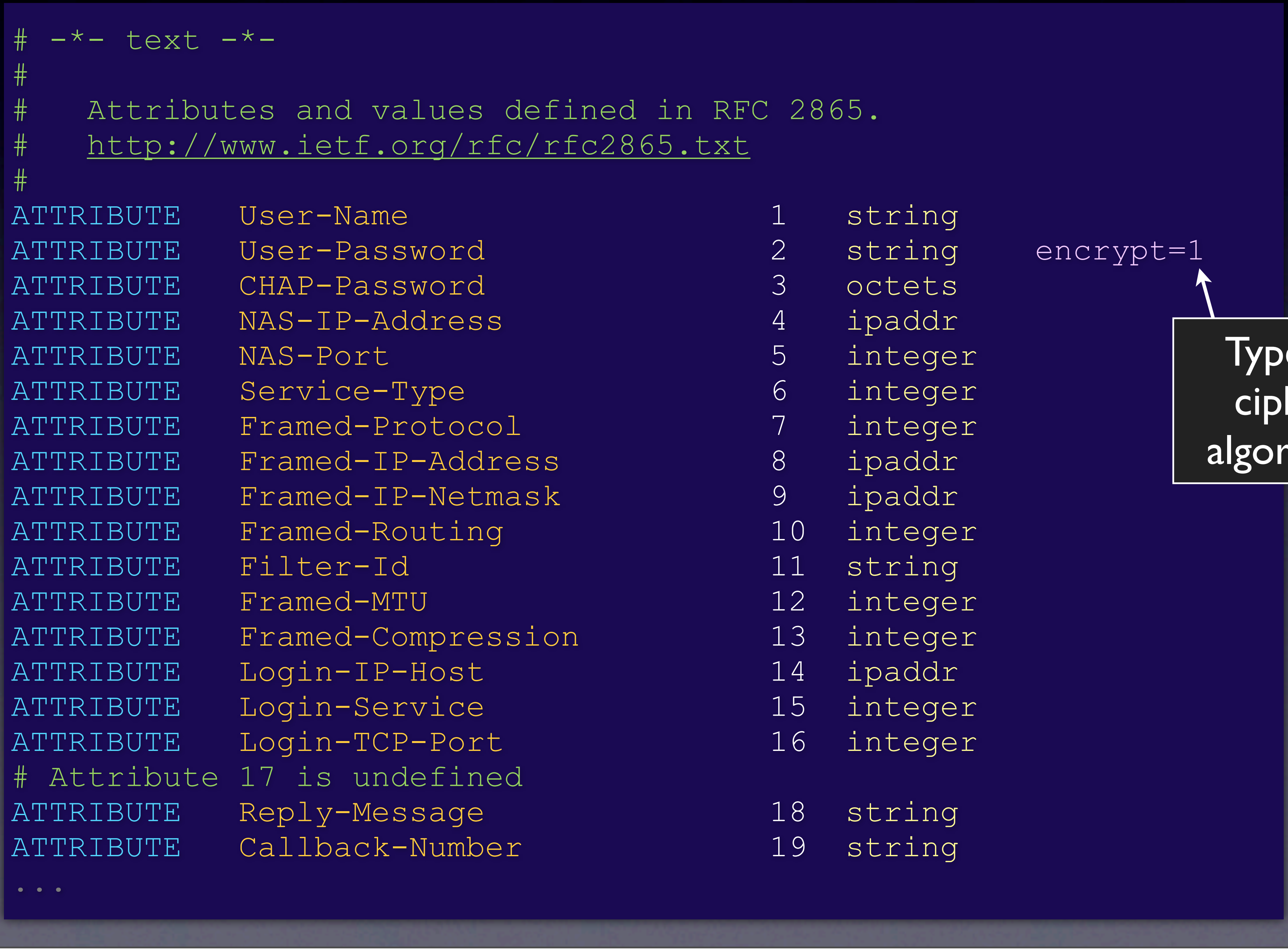
- The standard dictionary file simply includes many dictionaries:

```
$INCLUDE dictionary.rfc2865
$INCLUDE dictionary.rfc2866
$INCLUDE dictionary.rfc2867
...
$INCLUDE dictionary.cisco.bbsm
$INCLUDE dictionary.clavister
...
```

- If you wish to add attribute definitions for your own attributes, you should modify `/etc/freeradius/dictionary`, but *never* modify any `/usr/share/freeradius/dictionary.*`

The RADIUS dictionary

- For example, here is the beginning of the dictionary that defines the attributes of RFC 2865:

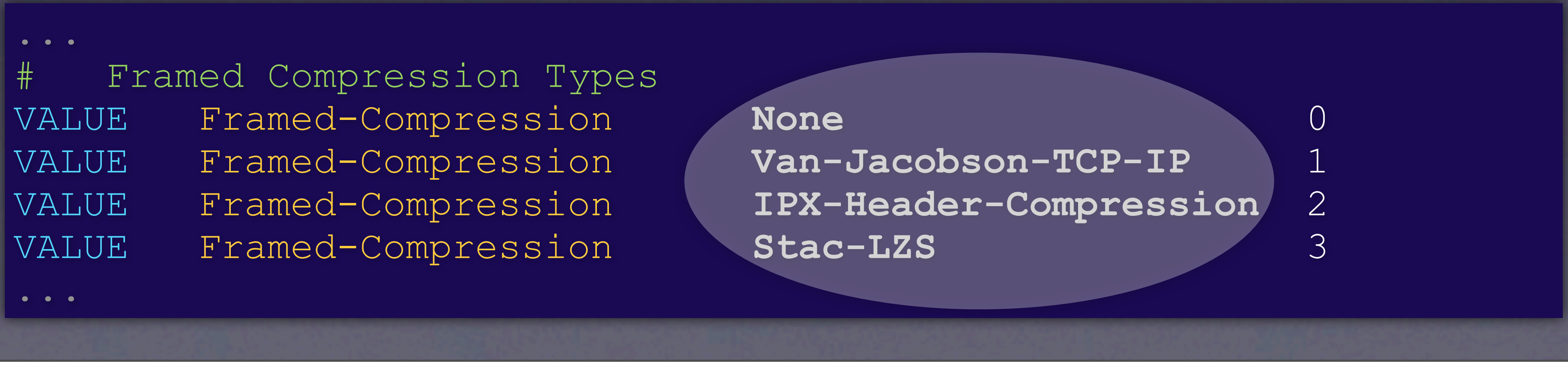


```
# -*- text -*-
#
#   Attributes and values defined in RFC 2865.
#   http://www.ietf.org/rfc/rfc2865.txt
#
ATTRIBUTE   User-Name                   1   string
ATTRIBUTE   User-Password               2   string    encrypt=1
ATTRIBUTE   CHAP-Password               3   octets
ATTRIBUTE   NAS-IP-Address              4   ipaddr
ATTRIBUTE   NAS-Port                    5   integer
ATTRIBUTE   Service-Type                6   integer
ATTRIBUTE   Framed-Protocol             7   integer
ATTRIBUTE   Framed-IP-Address           8   ipaddr
ATTRIBUTE   Framed-IP-Netmask           9   ipaddr
ATTRIBUTE   Framed-Routing              10  integer
ATTRIBUTE   Filter-Id                   11  string
ATTRIBUTE   Framed-MTU                  12  integer
ATTRIBUTE   Framed-Compression          13  integer
ATTRIBUTE   Login-IP-Host               14  ipaddr
ATTRIBUTE   Login-Service               15  integer
ATTRIBUTE   Login-TCP-Port              16  integer
# Attribute 17 is undefined
ATTRIBUTE   Reply-Message               18  string
ATTRIBUTE   Callback-Number             19  string
...
```

Type of cipher algorithm

The RADIUS dictionary

- For some attributes, the possible values are numbered, and what is actually sent in the RADIUS packets is that number (*not* the name of the value).
- The association between the name of the value and its number can be configured in the dictionary. You can then use the name instead of the number in the rest of the config.
- For example, `dictionary.rfc2865` contains the definition of the possible values for the `Framed-Compression` attribute (attribute number 13) :



```
...
#   Framed Compression Types
VALUE   Framed-Compression      None                    0
VALUE   Framed-Compression      Van-Jacobson-TCP-IP     1
VALUE   Framed-Compression      IPX-Header-Compression  2
VALUE   Framed-Compression      Stac-LZS                3
...
```

The RADIUS dictionary

- Finally, in the case of `Vendor-Specific` attributes, the vendor's number (assigned by the IANA) is sent in the RADIUS packets (*not* the vendor's name).

- Again, the dictionary allows you to associate each vendor's name with its number, so you can then use the vendor's name everywhere in the configuration, instead of its number

- For example, here's what Cisco's dictionary looks like, defined in `dictionary.cisco` (Cisco's IANA number is **9**):

```
VENDOR  Cisco  9

BEGIN-VENDOR  Cisco

ATTRIBUTE   Cisco-AVPair     1   string
ATTRIBUTE   Cisco-NAS-Port   2   string
...
VALUE   Cisco-Disconnect-Cause      Session-End-Callback    102
VALUE   Cisco-Disconnect-Cause      Invalid-Protocol        120

END-VENDOR  Cisco
```

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
  - Configuration of the RADIUS dictionary
  - Basic configuration of the server
  - Request management policies configuration
  - Modules configuration
  - Roaming configuration

Configuration syntax

- The file `/etc/freeradius/radiusd.conf` is the entry point for all freeRADIUS configuration (except for the dictionary configuration).
- Its syntax is fairly simple, it is just composed of:
  - variable definitions (ex: `prefix = /usr`)
  - module names (ex: `ldap`), alone on a line
  - and sections (ex : `authenticate { ... }`) which can contain all the above, as well as subsections (recursively)
  - ...plus comments, which can occur anywhere:
    `# a comment, up to the end of the line`

$INCLUDE

- You can include a file at any point in the configuration using the `$INCLUDE` keyword.
- You may also include a whole directory: all the files whose name only contains letters, numbers, dots (.), and underscores (_) will be included.
- This is how freeRADIUS's configuration is spread across many files, including all the files in **`/etc/freeradius/modules`** and **`/etc/freeradius/sites-enabled`**, as well as many files located in **`/etc/freeradius`**.
- This organization is a lot clearer than that of version 1.

The variables

- The values of the variables can be given with or without single or double quotes:

```
exec_prefix = /usr
exec_prefix = '/usr'   # is equivalent
exec_prefix = "/usr"   # again, equivalent
```

- The definition must fit on one line, or it must end with a backslash:

```
name = "my name is ve\
ry long" # name = "my name is very long"
```

- The value of a variable may be used later in the configuration to define another variable, using the syntax `${var}`:

```
sbindir = ${exec_prefix}/sbin
```

- This substitution only occurs upon freeRADIUS startup (there is no runtime performance cost)

The sections

- The syntax is simple:

```
name_of_the_section { # compulsory carriage return here
   ...
} # must be on its own line (not counting spaces and comments)
```

- In some predefined cases that we will see later, a second name may (or must) follow the first section name, for example:

```
...
authenticate {
   ...
   Auth-Type CHAP {
       ...
   }
   ...
}
...
```

radiusd.conf

- Here’s the start of the default content of `radiusd.conf`:

```
prefix = /usr
exec_prefix = /usr
sysconfdir = /etc
localstatedir = /var
sbindir = ${exec_prefix}/sbin
logdir = /var/log/freeradius
raddbdir = /etc/freeradius
radacctdir = ${logdir}/radacct
name = freeradius
confdir = ${raddbdir}
run_dir = ${localstatedir}/run/${name}
db_dir = ${raddbdir}
libdir = /usr/lib/freeradius
pidfile = ${run_dir}/${name}.pid
#chroot = /path/to/chroot/directory
```

Paths to the main directories and files (usually, they do not need to be changed)

```
user = freerad
group = freerad
```

Un*x user and group that the server will run as (should usually not be changed)

```
max_request_time = 30
cleanup_delay = 5
max_requests = 1024

#...
```

A few performance parameters that can be tweaked, depending on the load of the server (see the comments in `radiusd.conf` for more details)

`listen` sections

- By default, freeRADIUS listens on all the server’s IP addresses (that is, it listens on the wildcard address), and on the default RADIUS ports (which are 1812 for authentication and authorization, and 1813 for accounting)
- You may change this in the `listen` sections of `radiusd.conf`

```
#...
listen {
      type = auth
      ipaddr = 10.1.2.3
      port = 0    # zero means use standard port (1812 for auth)
}
listen {
      type = acct
      ipaddr = 10.1.2.3
      port = 2001 # here, we chose to use a non-standard port for accounting
}
#...
```

- You may add as many `listen` sections as needed

listen sections

- Possible options for a listen section:

```
listen {
      type = auth                          # Type of service (see below)
      ipaddr = *                           # For IPv4 (here we listen on all IP addresses)
#     ipv6addr = ::                        # For IPv6 (same as above, listen on all IPs)
      port = 0                             # Use the standard port for the service
#     interface = eth0                     # You may specify the interface to listen on
#     clients = per_socket_clients         # Only listen to requests from a list of clients
#     virtual_server = my_policy           # Handle requests using a specific policy handled
                                           # by a named virtual server (we willl come back
                                           # to this later)
```

- Here are the possible types of services:

```
#   auth    Authentification and authorization
#   acct    Accounting
#   proxy   Allows you to specify the source IP and source port used by the server
#           when it proxies requests to another RADIUS server
#   detail  Used to synchronize redundant RADIUS servers. This functionality replaces
#           the old «radrelay» daemon of version 1.
#   status  Listens to Status-Server requests, sent by the «radadmin» tool
#   coa     For CoA-Request and Disconnect-Request packets (see later)
```

`listen` sections

- See `sites-available/copy-acct-to-home-server` for an example that uses the `detail` type
- See `sites-available/status` for an example that uses the `status` type
- See `sites-available/originate-coa` for an example that uses the `coa` type

radiusd.conf (cont’d)

```
# ...

hostname_lookups = no
allow_core_dumps = no
regular_expressions     = yes
extended_expressions    = yes
```

Activate or deactivate reverse DNS (for logs), *core dumps* and regular expressions (see later)

```
security {
      max_attributes = 200
      reject_delay = 1
      status_server = no
}
```

Some counter-measures against a few well-known security attacks

```
thread pool {
      start_servers = 5
      max_servers = 32
      min_spare_servers = 3
      max_spare_servers = 10
      max_requests_per_server = 0
}
```

Threads management

```
log {
      destination = files
      file = ${logdir}/radius.log
      syslog_facility = daemon
      stripped_names = no
      auth = no
      auth_badpass = no
      auth_goodpass = no
}

# ...
```

Logs management

radiusd.conf (cont'd)

```
#...

checkrad = ${sbindir}/checkrad
```

This tool can query a NAS to check whether a user is connected or not

```
proxy_requests  = yes
$INCLUDE proxy.conf
```

Roaming configuration

```
$INCLUDE clients.conf
```

NAS configuration

```
modules {
      $INCLUDE ${confdir}/modules/
      $INCLUDE eap.conf
#     $INCLUDE sql.conf
#     $INCLUDE sql/mysql/counter.conf
#     $INCLUDE sqlippool.conf
}
```

Modules configuration

```
instantiate {
      exec
      expr
#     daily
      expiration
      logintime
}
```

Force instanciation of modules (see later)

```
$INCLUDE policy.conf

$INCLUDE sites-enabled/
```

Definitions of the virtual servers (they handle the requests) and their policies

clients.conf

- Configuration of all the NAS that will talk to the server

```
client localhost {
  ipaddr = 127.0.0.1
  secret = testing123
}
```

To run tests from the server itself

```
client meeting-room.wifi.wifirst.fr {
  shortname = wifi_meeting
```

The shortname is used to reference this NAS from the rest of the configuration. By default, it is the name stated at the beginning of the section.

```
  ipaddr = 10.1.9.4
# ipv6addr = ::
# netmask = 32
```

The NAS IP address or a subnet containing one or more NAS

```
  secret = "hEin/geo9c$be3Eet.ugh3le0eH"
```

An excellent secret is compulsory

```
  require_message_authenticator = yes
```

Defaults to «no». It is best to set this to «yes» if the NAS supports it.

```
  nastype = cisco
```

Used by the `checkrad` tool in order to known how to query the NAS

```
# virtual_server = politique_stricte
```

This allows a specific policy to be applied for this NAS

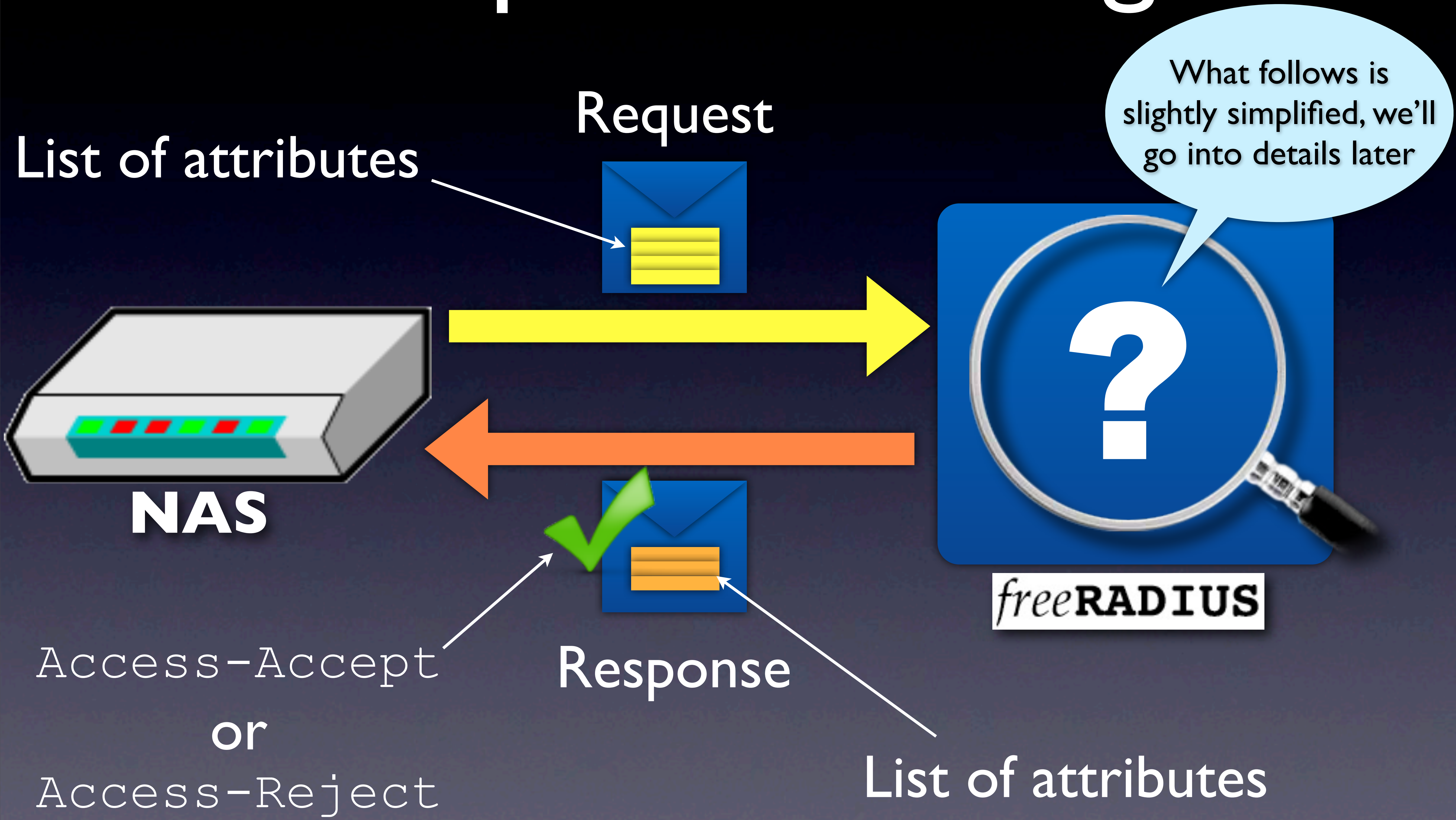
```
# coa_server = coa
}
```

CoA : see later

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
  - Configuration of the RADIUS dictionary
  - Basic configuration of the server
  - Request management policies configuration
  - Modules configuration
  - Roaming configuration

Request handling



What follows is slightly simplified, we'll go into details later

Request

List of attributes

NAS

freeRADIUS

Response

List of attributes

`Access-Accept`

or

`Access-Reject`

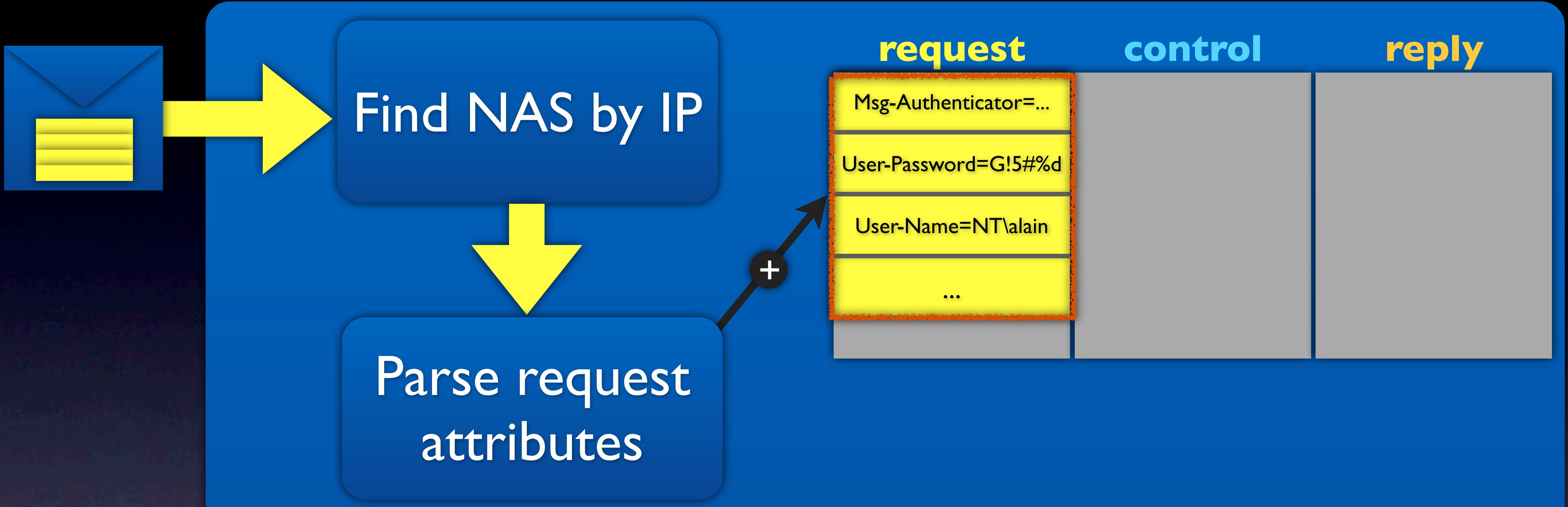
NAS lookup

Find NAS by IP

- A NAS is always looked up by the source IP address of the RADIUS packet
- If the NAS is not found, the packet is ignored
- All NAS configuration is loaded when freeRADIUS starts up, it is entirely static

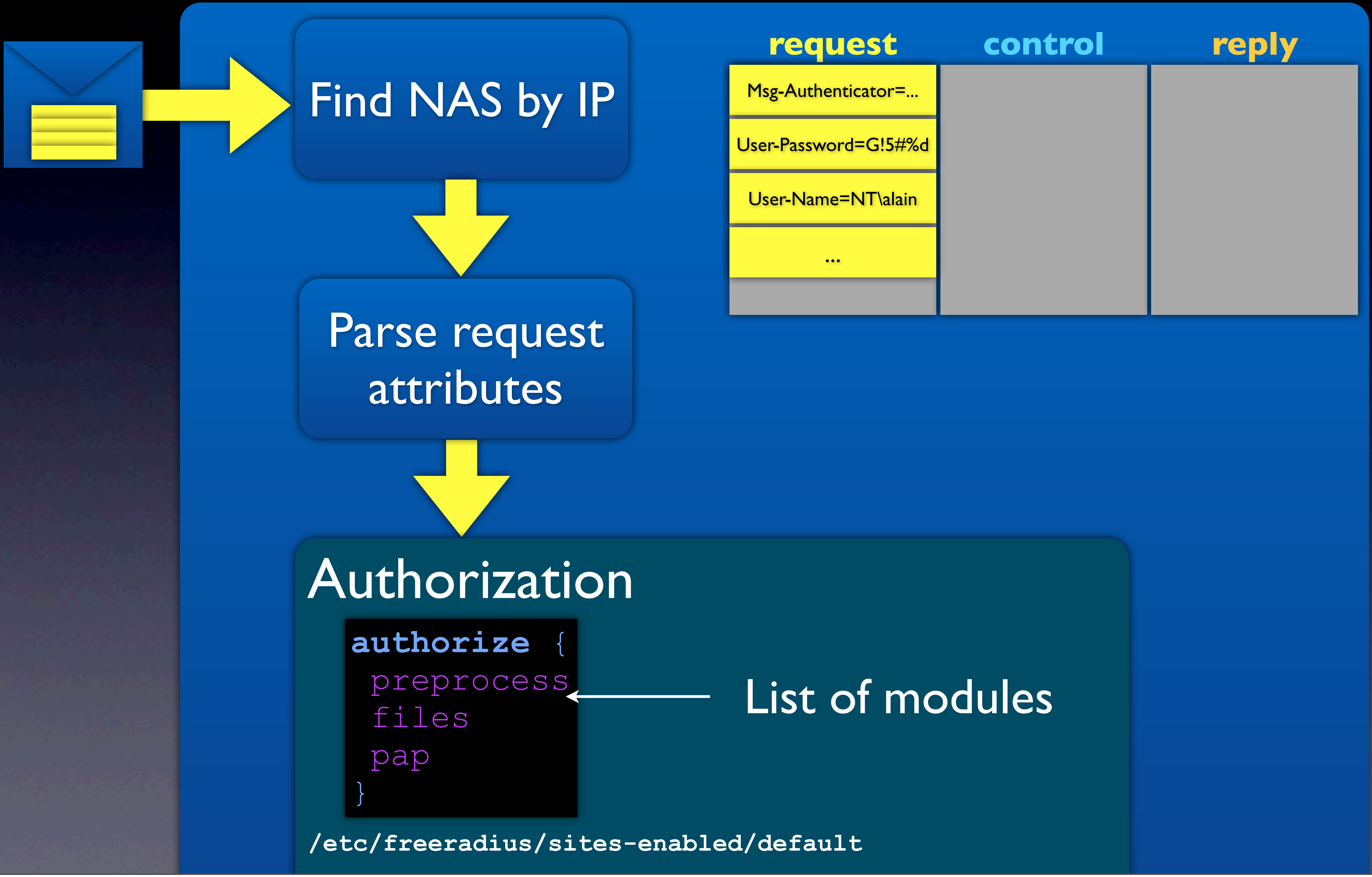
If you want to add, modify or delete a NAS, you need to restart freeRADIUS

Internal lists of attributes



- Notes:
  - The control attributes are sometimes called «config items»
  - There are a few other lists: proxy-request, proxy-response, outer.request, outer.reply, coa, etc.

Authorization phase



Find NAS by IP

request	control	reply
Msg-Authenticator=...		
User-Password=G!5#%d		
User-Name=NT\alain		
...		

Parse request attributes

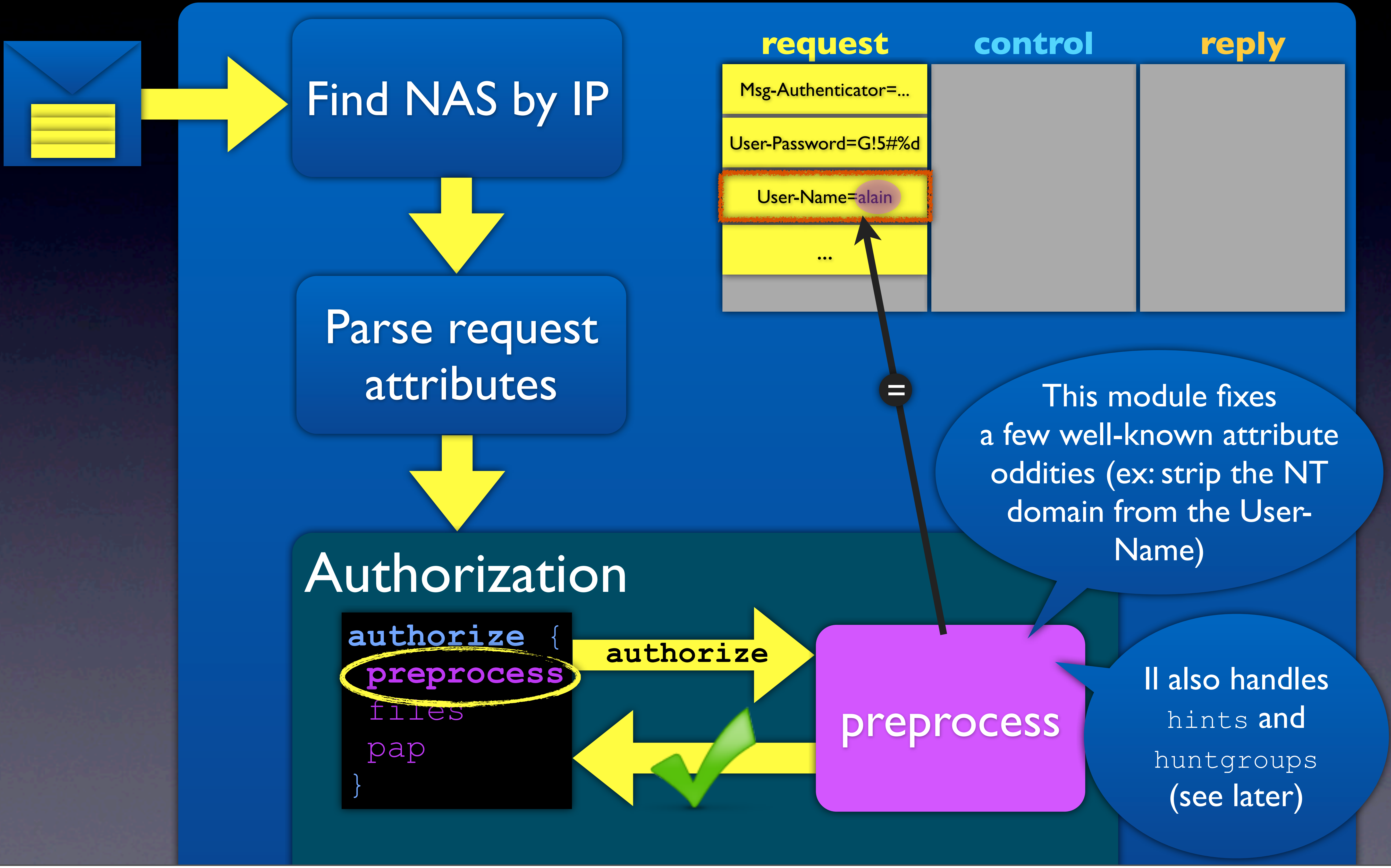
Authorization

```
authorize {
 preprocess
 files
 pap
}
```

List of modules

`/etc/freeradius/sites-enabled/default`

preprocess module



Find NAS by IP

Parse request attributes

Authorization

```
authorize {
  preprocess
  files
  pap
}
```

authorize

preprocess

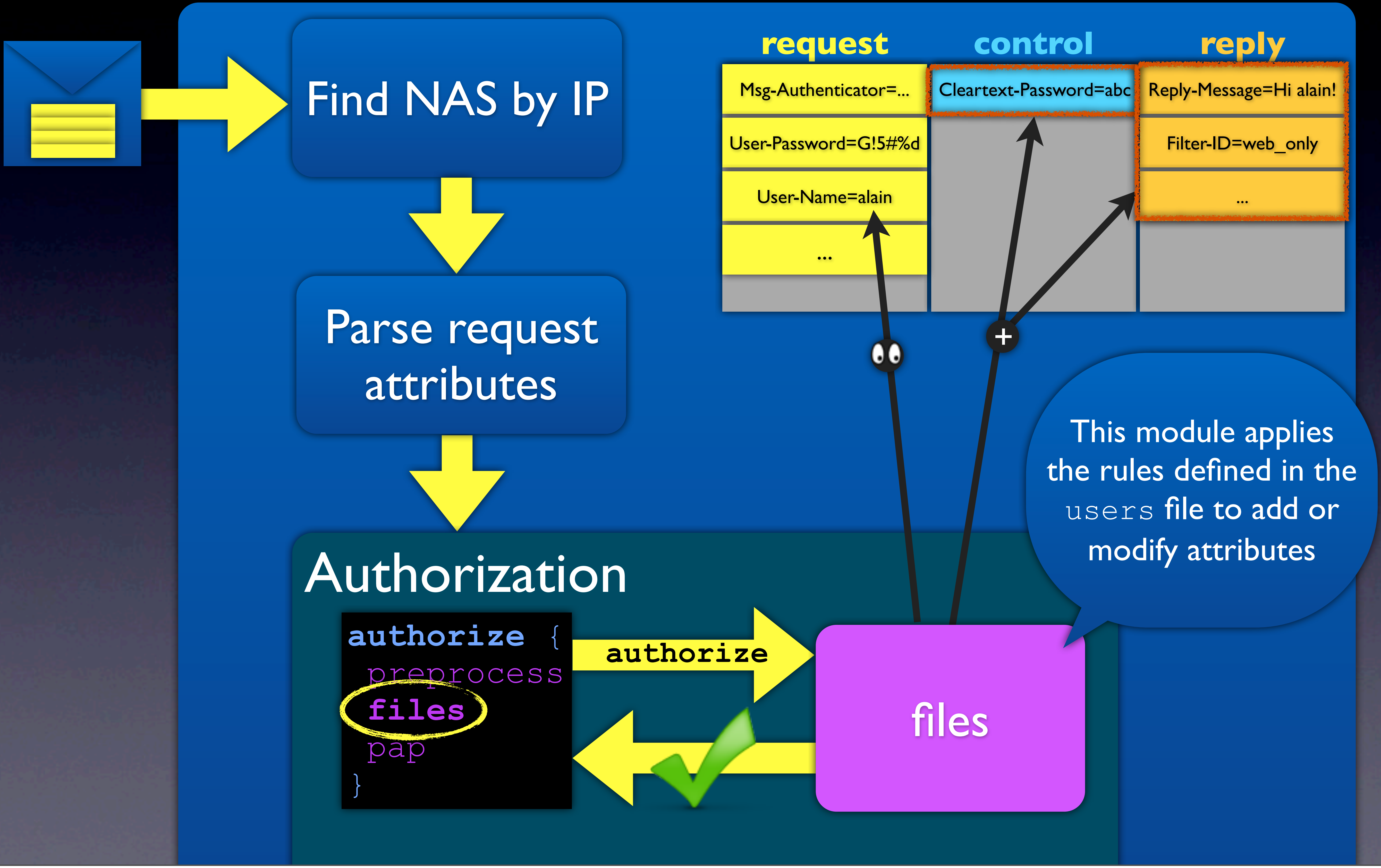
=

request	control	reply
Msg-Authenticator=...		
User-Password=G!5#%d		
User-Name=alain		
...		

This module fixes a few well-known attribute oddities (ex: strip the NT domain from the User-Name)

Il also handles hints and huntgroups (see later)

`files` module



request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d		Filter-ID=web_only
User-Name=alain		...
...		

```
authorize {
  preprocess
  files
  pap
}
```

This module applies the rules defined in the `users` file to add or modify attributes

One moment please!

The `files` and `preprocess` modules are important... let’s look at them a little closer before we come back to the request handling logic

users file

- The files module reads the /etc/freeradius/users file which contains rules to add, delete or modify attributes in the **control** and **reply** attributes lists
- This file is composed of a list of rules, each having the following format:

```
login condition1, condition2, ..., control_operation1, control_operation2,...
        reply_operation1,
        reply_operation2,
        ...
```

Tab (not spaces)

one reply operation per line

all control operations on the first line

- Example :

```
alain Huntgroup-Name == "switch7_ports_1_a_12", Cleartext-Password := "abc"
        Reply-Message = "Hi alain!",
        Filter-ID = "web_only"
```

Do not forget the commas

`users` rules processing

- The file is read in order, until a rule is found whose `login` field matches the user’s login (from the `User-Name` attribute) **and** whose `conditions` are all met (or else, freeRADIUS just continues to try to find a matching rule)

- As soon as a matching rule is found, its operations are executed: attributes are added, deleted or modified in the `control` and/or `reply` lists, then the module exits

- Note: in the freeRADIUS documentation, the `conditions` and the operations on the `control` list attributes are both called «*check items*»

Conditions format

- Each condition applies to an attribute in the `request` list (or the `control` list if it is not found in the `request` list)
- The conditions are formatted as follows:
  `attribute operator value`
- The possible operators are:

`==`	equal to
`!=`	not equal to
`>`	strictly greater than (only for integer attributes)
`>=`	greater or equal to (only for integer attributes)
`<`	strictly lower than (only for integer attributes)
`<=`	lower or equal to (only for integer attributes)
`=~`	matches the regular expression (only for string and text attributes)
`!~`	does not match the regular expression (only for string and text attributes)
`=*`	is present (the value is ignored, you can write `attr =* yes` for example)
`!*`	is not present (again, the value is ignored)

Operations format

- To add or modify an attribute in the `control` or `reply` lists, the syntax is once again:
  `attribute operator value`

- The possible operators are:

`=` adds the attribute set to the given value, or does nothing if the attribute already exists

`:=` adds the attribute set to the given value, overwriting the existing value if the attribute already exists

`+=` adds the attribute set to the given value, even if it already exists (the same attribute may then appear multiple times in the RADIUS packet)

- **Warning**: don't confuse `=`, `:=` and `==`
  It must be clear to you that the operations (`=`, `:=` and `+=`) are only executed if the `login` matches and if the conditions (`==`, `>=`...) are all met

Fall-Through attribute

- By default, as soon as the `files` module finds a matching rule, it applies its operations then exits
- But you can tell it to continue to go through the rules, simply by adding **`Fall-Through=Yes`** at the end of the rule, after the end of the list of **`reply`** attributes (note that this attribute is not added to the **`reply`** attributes list)
- Example:

```
alain Cleartext-Password := "abc"
        Reply-Message = "Hi alain!",
        Fall-Through = Yes

alain Huntgroup-Name == "switch7_ports_1_to_12"
        Filter-ID = "web_only"

alain Huntgroup-Name == "switch7_ports_13_to_24"
        Filter-ID = "voip_only"
```

`alain`'s passord is always «abc» and the welcome message is always «Hi alain!»

Condition on the location and connection port (see later)

Filtering that the NAS should apply

DEFAULT login

- If you write **DEFAULT** instead of the user's **login**, then the rule is applied for all users, provided the conditions are all met
- Example:

```
DEFAULT
        Reply-Message = "Hi %{User-Name}!",
        Fall-Through = Yes

DEFAULT Huntgroup-Name == "switch7_ports_1_a_12"
        Filter-ID = "web_only",
        Fall-Through = Yes

DEFAULT Huntgroup-Name == "switch7_ports_13_a_24"
        Filter-ID = "voip_only",
        Fall-Through = Yes

alain  Cleartext-Password := "abc"

pierre Cleartext-Password := "def"

jean   Cleartext-Password := "ghi"

DEFAULT Auth-Type := Reject
        Reply-Message = "No way! Unknown user."
```

This will be dynamically substituted by the value of the **User-Name** attribute

Every user will have a personnalized welcome message including his own login

For the users who connect to ports 1 to 12 on switch 7, the access will be limited to the Web

And on ports 13 to 24, the access will be limited to VoIP

If this rule is reached, it means that the user is unknown: we reject him, and overwrite the reply message defined above

Translations

- The syntax `%{attribute}` is dynamically substituted (this is called *translation* or *xlat*) by the value of the attribute

  ➡ Example : `"abc%{User-Name}def"` will be translated to `"abcjoedef"` for user `joe`

- This should not be confused with the `${variable}` syntax that we saw earlier, which is only expanded upon startup

- You may also use: `"%{%{attribute}:-default_value}"` which is translated to the value of the attribute, or to the string `"default_value"` if the attribute is not defined

- You may use this syntax recursively, for example: `"%{%{Stripped-User-Name}:-%{%{User-Name}:-unknown}}"`

Will be translated to the value of `Stripped-User-Name` or, if it is undefined, to the value of `User-Name` or, if it is undefined, to `unknown`

Translations

- Some modules offer an xlat function which can be called with the following syntax: `%{module_name:parameters}`
- Modules `sql`, `ldap`, `expr`, `exec` and `perl` have an xlat function. Here are a few examples:
    - `%{sql:select credit from credits where login='%{User-Name}'}`
    - `%{ldap:ldap:///dc=company,dc=com?uid?sub?uid=%u}`
    - `%{expr:2*%{Session-Timeout}+10}`
    - `%{exec:/usr/bin/mon_prog %{User-Name} %{Session-Timeout}}`
    - `%{perl:%{User-Name} %{Session-Timeout}}`

Calls a personnalizable perl function

- The parameters can themselves contain substitutions: the module evaluates the translation (possibly by calling another module's xlat function) then escapes the result if necessary. For example, if the `User-Name` is `joe's`, then the `sql` module will substitute `%{User-Name}` by `joe=27s` (MIME encoding) before running the SQL query.

The *huntgroups*

- A *huntgroup* is a set of locations and/or connection ports (the `preprocess` module must be activated in order to use *huntgroups*)
- You may then filter on locations and ports in the `users` file by applying a condition like **`Huntgroup-Name == "..."`**
- Huntgroups are defined in `/etc/freeradius/huntgroups`
- For example:

```
switch7_ports_1_to_12  NAS-IP-Address == 10.4.3.2, NAS-Port-Id == 1-12
switch7_ports_13_to_24 NAS-IP-Address == 10.4.3.2, NAS-Port-Id == 13-24
switchs_1_to_3 NAS-IP-Address == 10.4.3.2
switchs_1_to_3 NAS-IP-Address == 10.4.3.3
switchs_1_to_3 NAS-IP-Address == 10.4.3.4
ports_voip NAS-IP-Address == 10.4.3.2, NAS-Port-Id == 13-24
ports_voip NAS-IP-Address == 10.4.3.3, NAS-Port-Id == 1,3-7,9
ports_voip NAS-IP-Address == 10.4.3.4, NAS-Port-Id == 1,10-15
```

A *huntgroup* can be composed of multiple NASes

Or even multiple sets of ports on different NASes

The *hints*

- The `preprocess` module also allows you to define *hints*: these are prefixes or suffixes that the user can add to his login in order to indicate what service he wishes
- They are defined in `/etc/freeradius/hints` using a format identical to that of file `/etc/freeradius/users`
- For example:

```
DEFAULT Suffix == ".ppp", Strip-User-Name = Yes
    Hint = "PPP",
    Service-Type = Framed-User,
    Framed-Protocol = PPP
```

The **`User-Name`** attribute will be modified in the request to remove the «`.ppp`» suffix

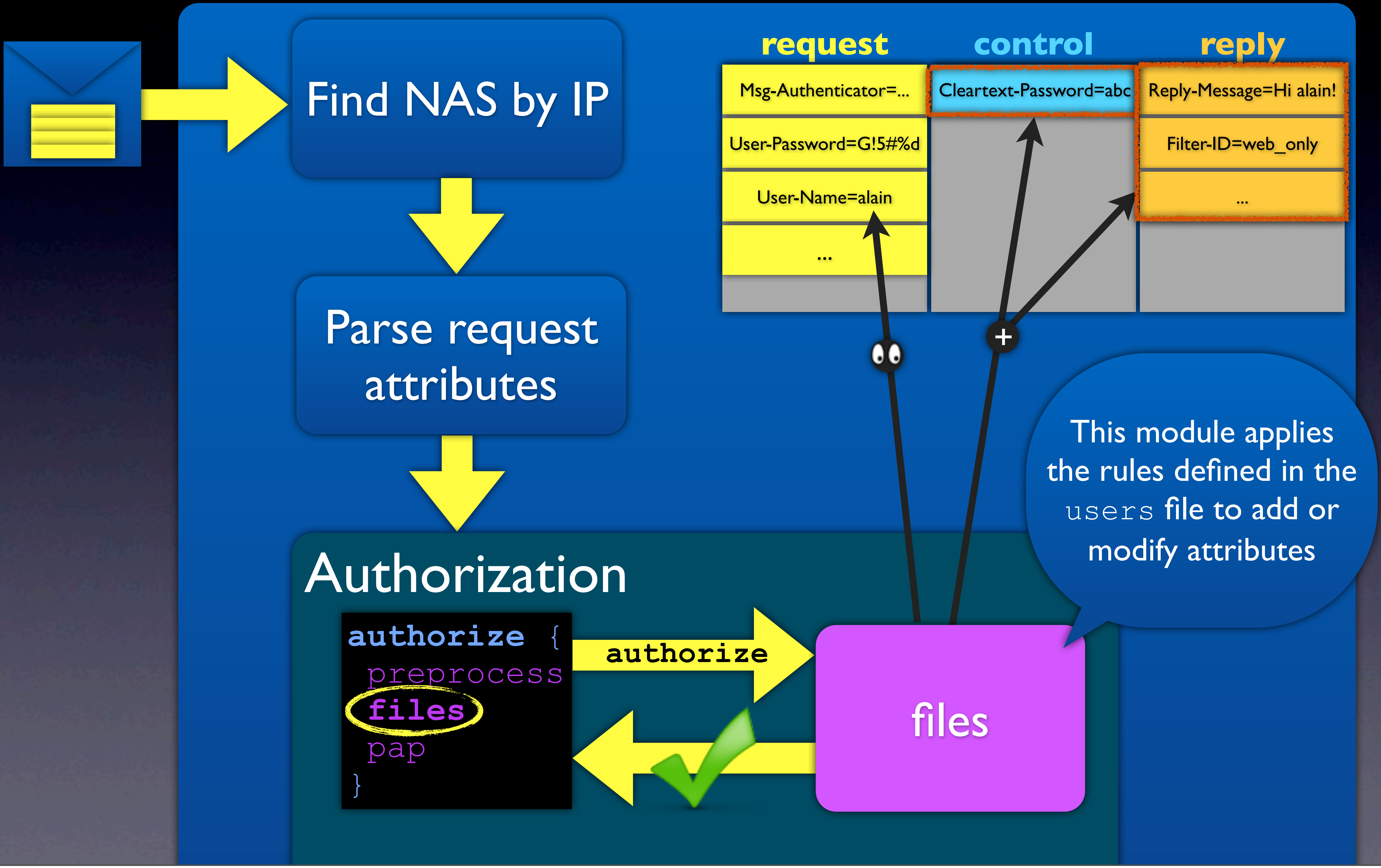
- In this example, if the **`User-Name`** attribute ends with "**`.ppp`**" then the attributes **`Hint`**, **`Service-Type`** and **`Framed-Protocol`** will be added to the **`request`** internal list

Warning: unlike the `users` file, the `hints` file will modify the request!

Let’s continue...

The request had gone through the `preprocess` module, and was now being handled by the `files` module in the `authorize` section

files module



Find NAS by IP

Parse request attributes

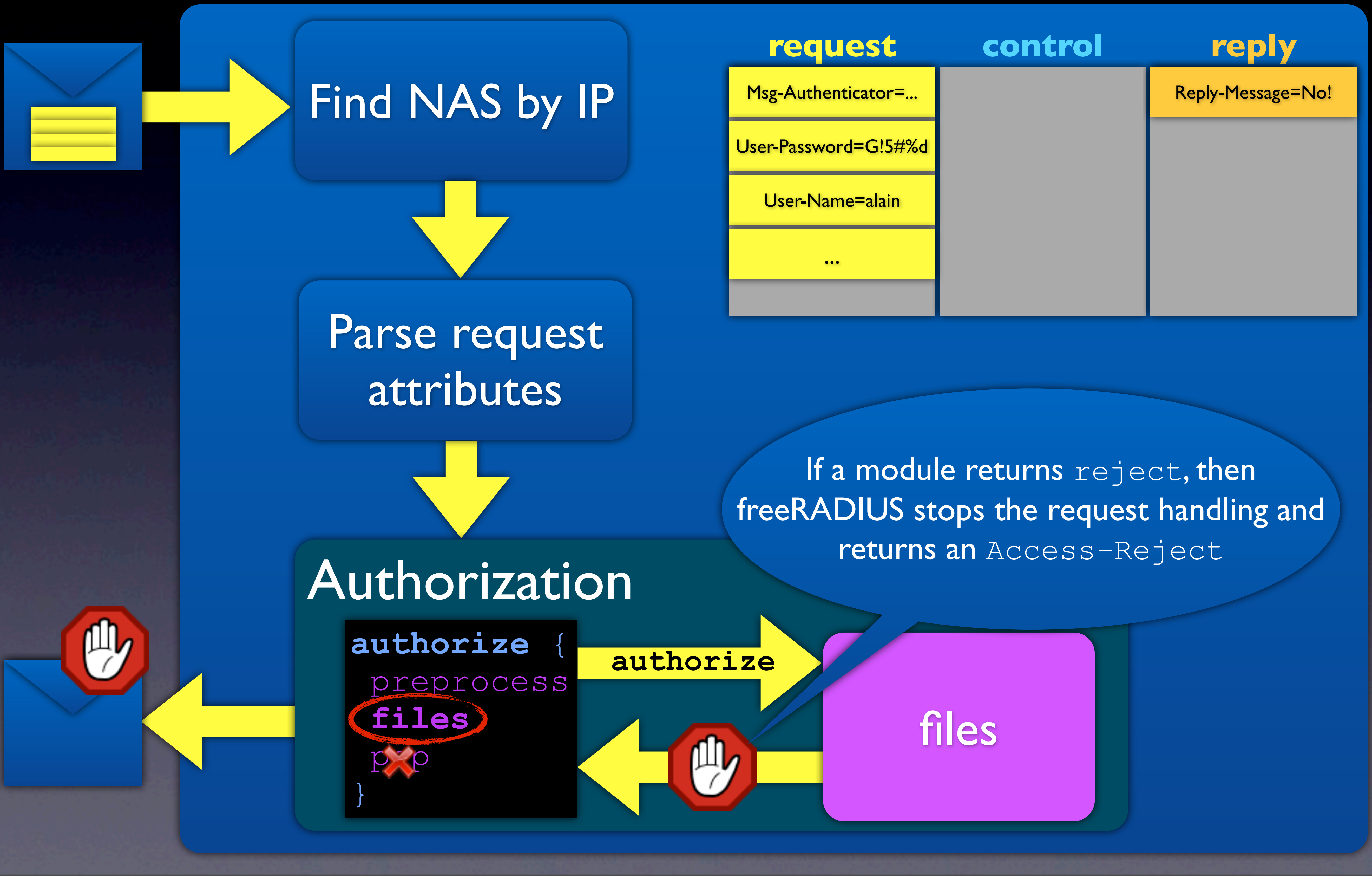
request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d		Filter-ID=web_only
User-Name=alain		...
...		

Authorization

```
authorize {
 preprocess
 files
 pap
}
```

This module applies the rules defined in the users file to add or modify attributes

Rejecting a user

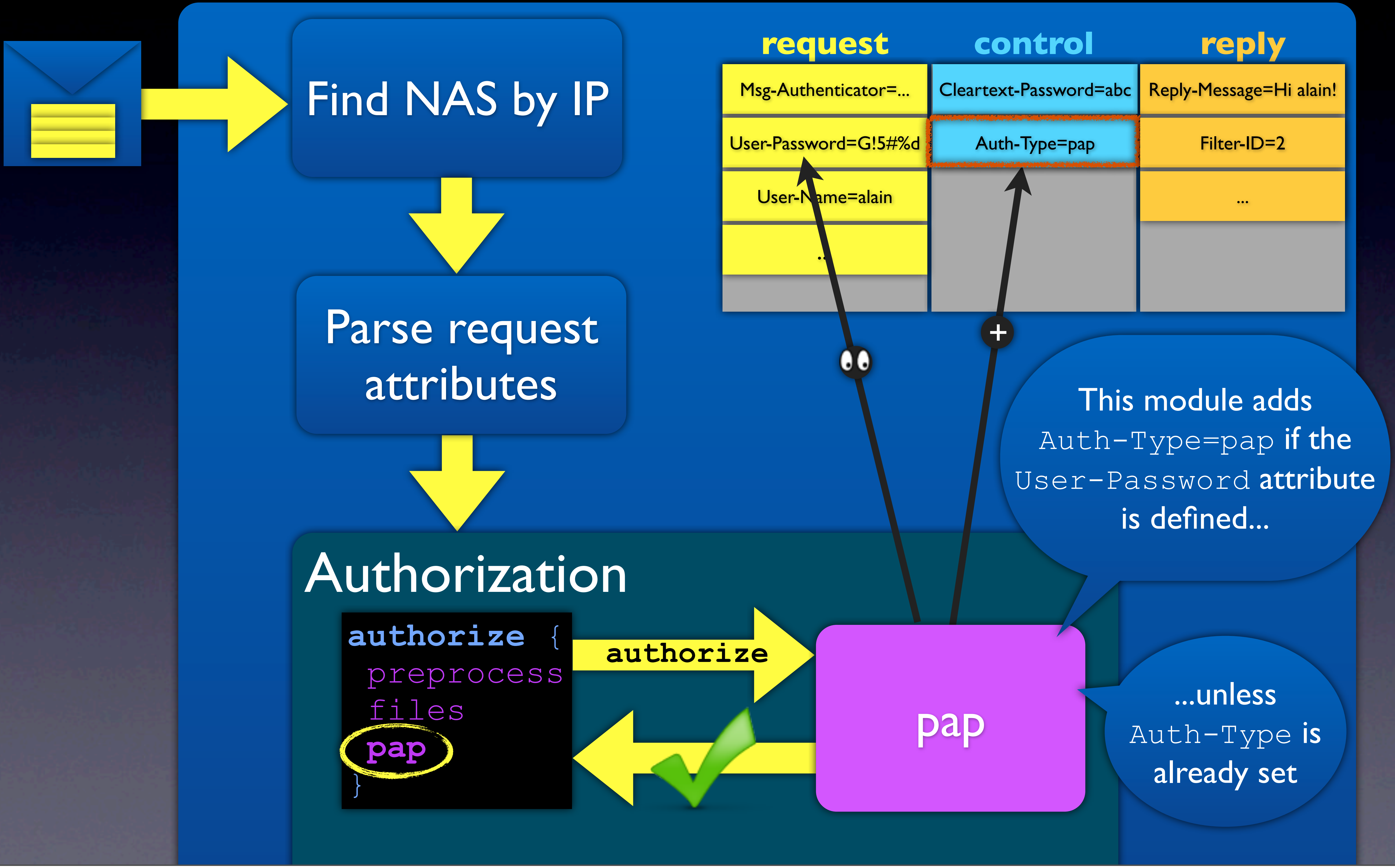


request	control	reply
Msg-Authenticator=...		Reply-Message=No!
User-Password=G!5#%d		
User-Name=alain		
...		

```
authorize {
  preprocess
  files
  pap
}
```

If a module returns `reject`, then freeRADIUS stops the request handling and returns an `Access-Reject`

pap module



request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d	Auth-Type=pap	Filter-ID=2
User-Name=alain		...
...		

```
authorize {
  preprocess
  files
  pap
}
```

This module adds `Auth-Type=pap` if the `User-Password` attribute is defined...

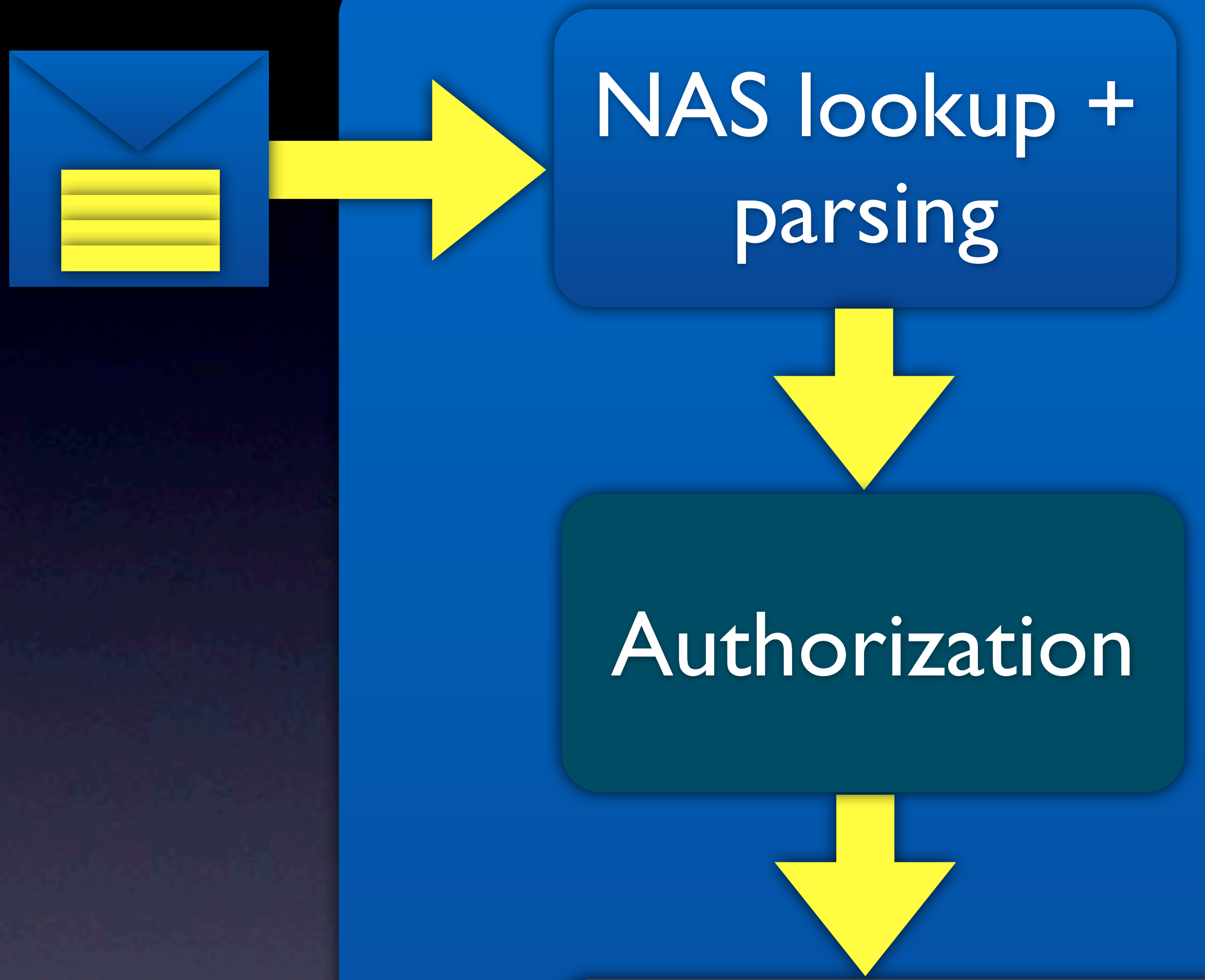
...unless `Auth-Type` is already set

`Autz-Type` subsection

- In the `authorize` section, subsections named «`Autz-Type XXX`» may be defined
- The `authorize` section is first executed without those subsections
- If the `Autz-Type` attribute is defined after the execution of the `authorize` section, and if the user was not rejected, then the `Autz-Type` subsection corresponding to the value of the `Autz-Type` attribute is executed alone
- This allows a specific authorization policy to be chosen dynamically

`Autz-Type` = **Auth**orization Type
`Auth-Type` = **Auth**entication Type

Authentication



NAS lookup + parsing

Authorization

request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d	Auth-Type=pap	Filter-ID=2
User-Name=alain		...
...		

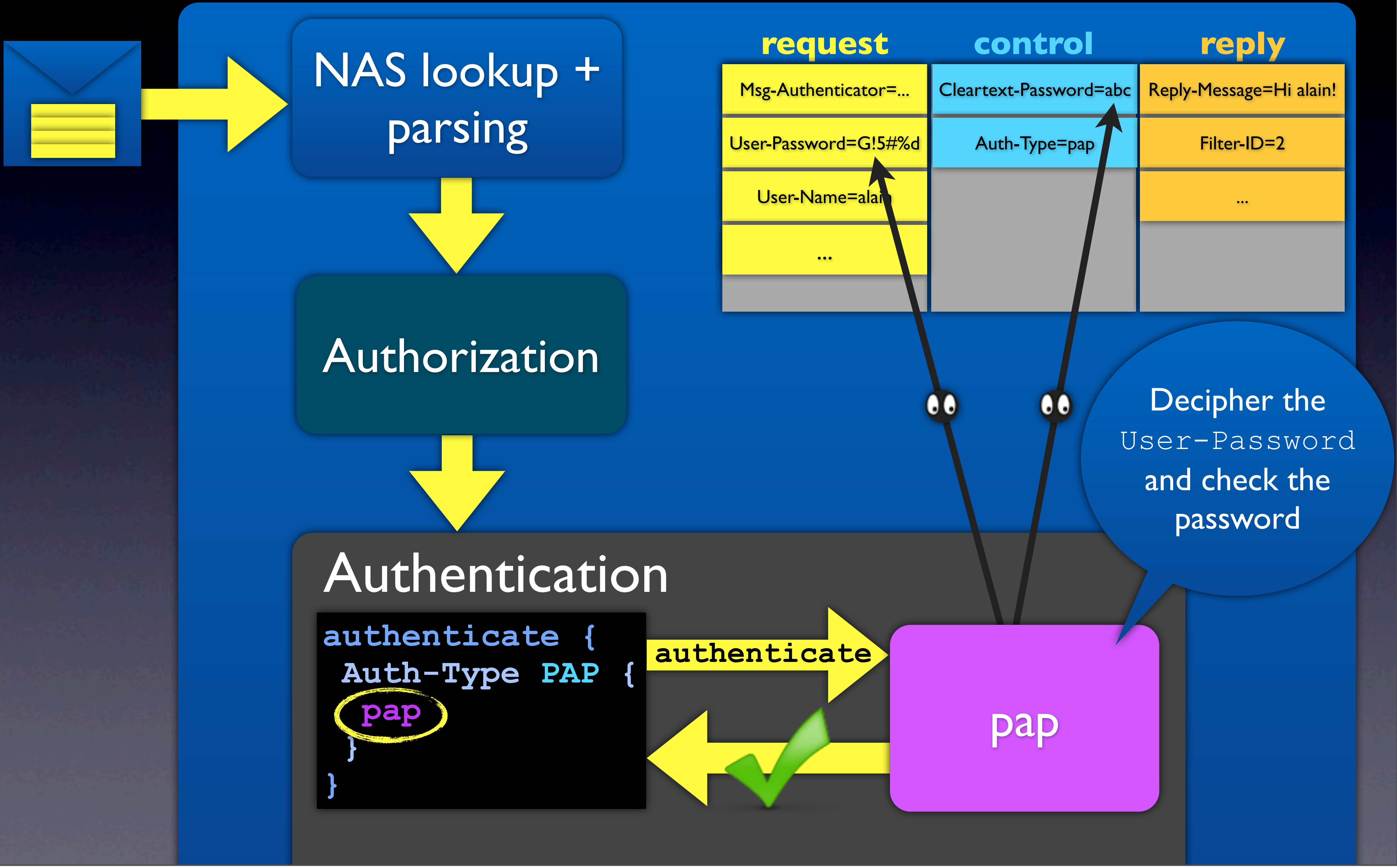
Authentication

```
authenticate {
 Auth-Type PAP {
  pap
 }
}
```

/etc/freeradius/sites-enabled/default

If the `Auth-Type` attribute is defined and if a corresponding subsection is defined, then it is executed (alone), or else the `authenticate` section is executed (without its subsections).

pap module (again)

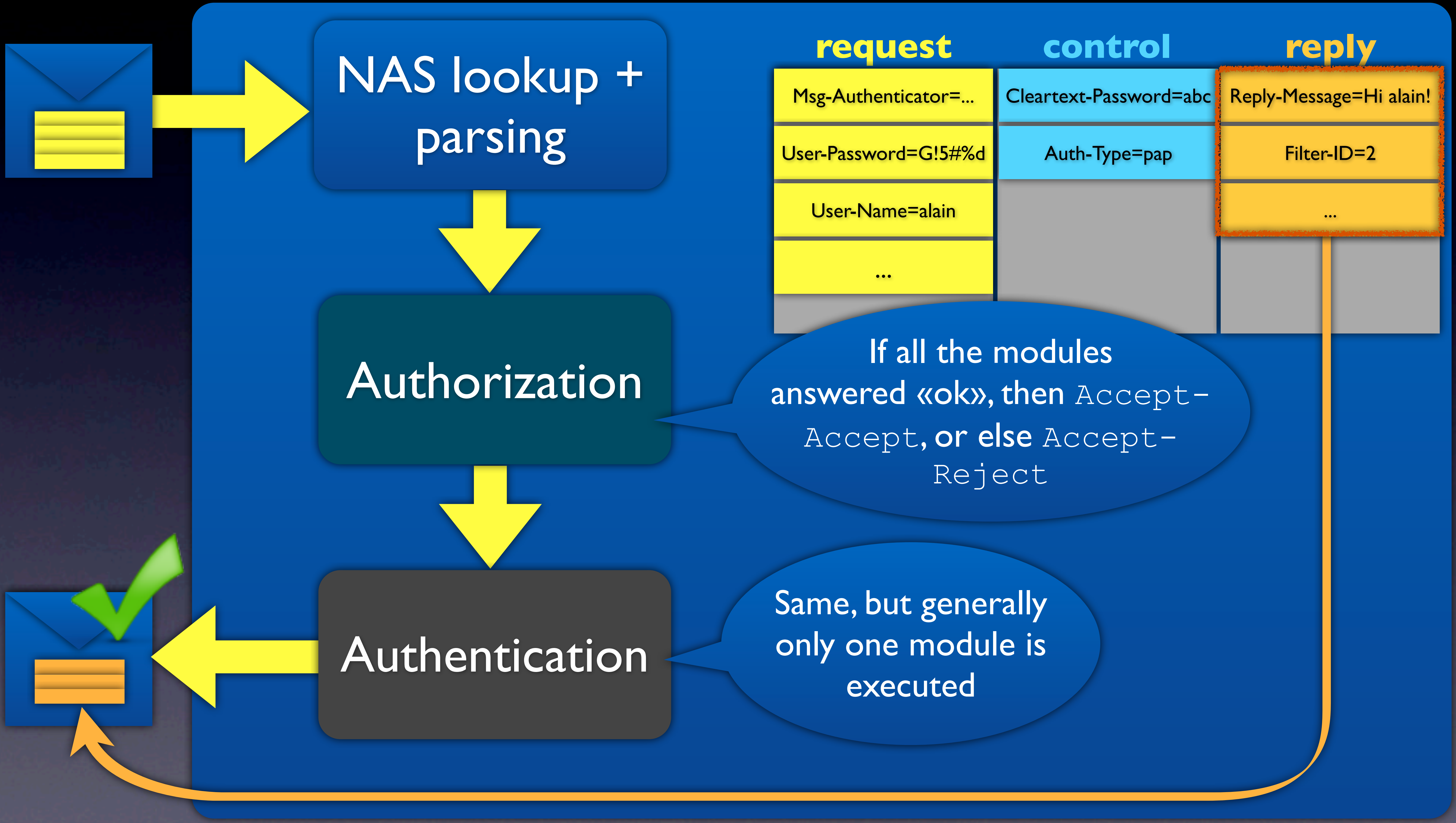


request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d	Auth-Type=pap	Filter-ID=2
User-Name=alain		...
...		

```
authenticate {
 Auth-Type PAP {
  pap
 }
}
```

Decipher the User-Password and check the password

Response at last!



request	control	reply
Msg-Authenticator=...	Cleartext-Password=abc	Reply-Message=Hi alain!
User-Password=G!5#%d	Auth-Type=pap	Filter-ID=2
User-Name=alain		...
...		

If all the modules answered «ok», then `Accept-Accept`, or else `Accept-Reject`

Same, but generally only one module is executed

*free*RADIUS

Authentication data

- Depending on the chosen authentication method, different types of data are sent by the NAS to the RADIUS server, for example:
  - PAP : user password ciphered using the RADIUS secret
  - CHAP : MD5 hash of the password + challenge + CHAP-ID
  - EAP/MD5 : similar to CHAP
  - TTLS/PAP : user password ciphered within a TLS tunnel
  - PEAP/MS-CHAP-v2 : hash of a NT hash within a TLS tunnel
  - EAP/TLS : user's TLS certificate
  - ...

Password verification

- If you enter the users' cleartext passwords in the `users` file (example: `Cleartext-Password:="a3d$G4"`) then freeRADIUS can check user passwords easily:
  - for PAP and TTLS/PAP: the received ciphered password is deciphered and simply compared to the cleartext password available in the `users` file
  - for CHAP and EAP/MD5: freeRADIUS calculates the MD5 hash of the user's password from the `users` file + the received challenge and CHAP-ID, and compares the result to the received MD5 hash
  - for PEAP/MS-CHAP-v2: freeRADIUS applies the MS-CHAP-v2 algorithm to calculate the appropriate hash using the user's password from the `users` file and the data received in the RADIUS request (MS-CHAP-v2 challenge), and compares the result to the received hash

Cleartext password?

For trivial security reasons, it is strongly recommanded *not* to store the users' passwords in cleartext

Password storage

- In the `users` file, it is possible to store a hash of each password instead of the cleartext passwords:
  - `Crypt-Password` : Unix crypt password
  - `MD5-Password` : MD5 hash
  - `SMD5-Password` : MD5 hash of the password + salt
  - `SHA-Password` : SHA1 hash
  - `SSHA-Password` : SHA1 hash of the password + salt
  - `NT-Password` : Windows NT hash
  - `LM-Password` : Windows Lan Manager hash

Hash incompatibilities

- Unfortunately, a hash is by definition a one-way function, meaning that it is impossible to guess the password if you know only its hash (unless you try all possible passwords)
- Therefore, if you store the SHA1 hash of the users' passwords, you will not be able to use the CHAP authentication method, for example, because freeRADIUS will have no way of knowing if the SHA1 hash stored in the `users` file and the received MD5 hash have been calculated from the same password or not
- As a matter of fact, since the MD5 hash that is transmitted when using the CHAP authentication method is *not* a hash of the user's password alone (but a hash of the password plus a challenge and a CHAP-ID), you cannot use CHAP authentication if you chose to store MD5 hashes of the user passwords!

Compatibility table

- The following table shows, for each authentication method, the compatible password storage formats:

Storage Method	Clear	Crypt	MD5	SHA1	SMD5	SSHA1	NT	LM
PAP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CHAP	Yes	No	No	No	No	No	No	No
EAP/MD5	Yes	No	No	No	No	No	No	No
TTLS/PAP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PEAP/MS-CHAP-v2	Yes	No	No	No	No	No	Yes	Yes

Safe methods?

If a hacker gets access to the exchanges between the NAS and the RADIUS server, then not only will he have access to the unciphered data of the RADIUS packets, but he can go further, depending on the authentication method used:

- The PAP method is quite unsafe, because it is both vulnerable to offline dictionary attacks and to replay attacks

➡ Offline dictionary attack: a hacker enters a random password, then captures the corresponding RADIUS exchange. He can then try millions of RADIUS secrets until he finds one that produces the same ciphered password. Since he now has the RADIUS secret, he can decipher all the passwords from then on.

➡ Replay attack: the hacker captures a successful RADDIUS exchange and simply repeats it later on. He can connect with someone else's identity (without having to know his password).

Safe methods?

- The CHAP method is also vulnerable to offline dictionary attacks and replay attacks. But when the hacker manages to find the RADIUS secret, he can only decipher the passwords of users who use the PAP method (and he can also decipher all attributes that were ciphered using the RADIUS secret).
- The EAP/MD5 method has the same issues as CHAP.
- Methods that rely on a TLS tunnel are immune to offline dictionary attacks and replay attacks: they are therefore much safer.
- Their only problem is that they require the user to check the server's certificate... and many users simply don't bother to do so.

It is safer to use PEAP or TTLS

EAP/TLS, EAP/SIM, EAP/GTC

- The EAP/TLS does not transmit any password: during authentication, the user checks the server's certificate, and the server checks the user's certificate

  ➡ This method is immune to offline dictionary attacks, as well as *online* dictionary attacks (since no password is exchanged) and it is also immune to replay attacks. It is therefore even safer than PEAP and TTLS...

  ➡ ...but it is a bit tedious to implement because a TLS certificate needs to be installed on every user's system

- A higher degree of security can still be achieved using security cards, because the user possesses both a physical object (the card) and a secret (the PIN code): this is called a dual-factor authentication (2FA). This requires a complex hardware and software infrastructure, hence this solution is usually limited to telco operators and large enterprises. For security cards, the EAP method used is either EAP/SIM (for telco operators) or EAP/GTC (for other security cards)

Compatibility table

- Let’s amend the compatibility table, for more security:

<table>
<tr><th>Storage
Method</th><th>Clear</th><th>Crypt</th><th>MD5</th><th>SHA1</th><th>SMD5</th><th>SSHA1</th><th>NT</th><th>LM</th></tr>
<tr><td>PAP</td><td rowspan="5">Unsafe storage</td><td rowspan="3" colspan="7">Unsafe methods</td></tr>
<tr><td>CHAP</td></tr>
<tr><td>EAP/MD5</td></tr>
<tr><td>TTLS/PAP</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td><td>Yes</td></tr>
<tr><td>PEAP/MS-CHAP-v2</td><td>No</td><td>No</td><td>No</td><td>No</td><td>No</td><td>Yes</td><td>Yes</td></tr>
</table>

Module return codes

Up to now, we have assumed that a module either answered «**success**» or «**failure**», and in case of a failure, the request handling process would stop immediately. In fact, each module can return any one of the following codes:

Code	Meaning	Action
`notfound`	User was not found	**Continue**
`noop`	Module is not applicable (it did nothing)	**Continue**
`ok`	User accepted	**Continue**
`updated`	User accepted and attribute list updated	**Continue**
`fail`	Module failed (ex: database access failure)	**Stop + Reject**
`reject`	User rejected	**Stop + Reject**
`userlock`	User rejected because his account is locked	**Stop + Reject**
`invalid`	User rejected because his configuration is invalid	**Stop + Reject**
`handled`	Module handled the request and its response (if any)	**Stop + nothing**

Return codes priorities

- What should be done, for example, if a module answers `noop` in the `authorize` section, then the following module answers `notfound`, and the last module in the section answers `noop`? Logically, the result of the `authorize` section will be `notfound` (and freeRADIUS will reject the user).

- If one of those modules had answered `ok`, the result of the section would have been `ok` (and freeRADIUS would have continued on with the request handling process in the `authenticate` section)

- In conclusion, if no module returns an immediate failure, then a priority scale has to be applied between the module return codes in order to determine what the result of the section is

- By default, the priority scale is:
  `updated` > `ok` > `notfound` > `noop`

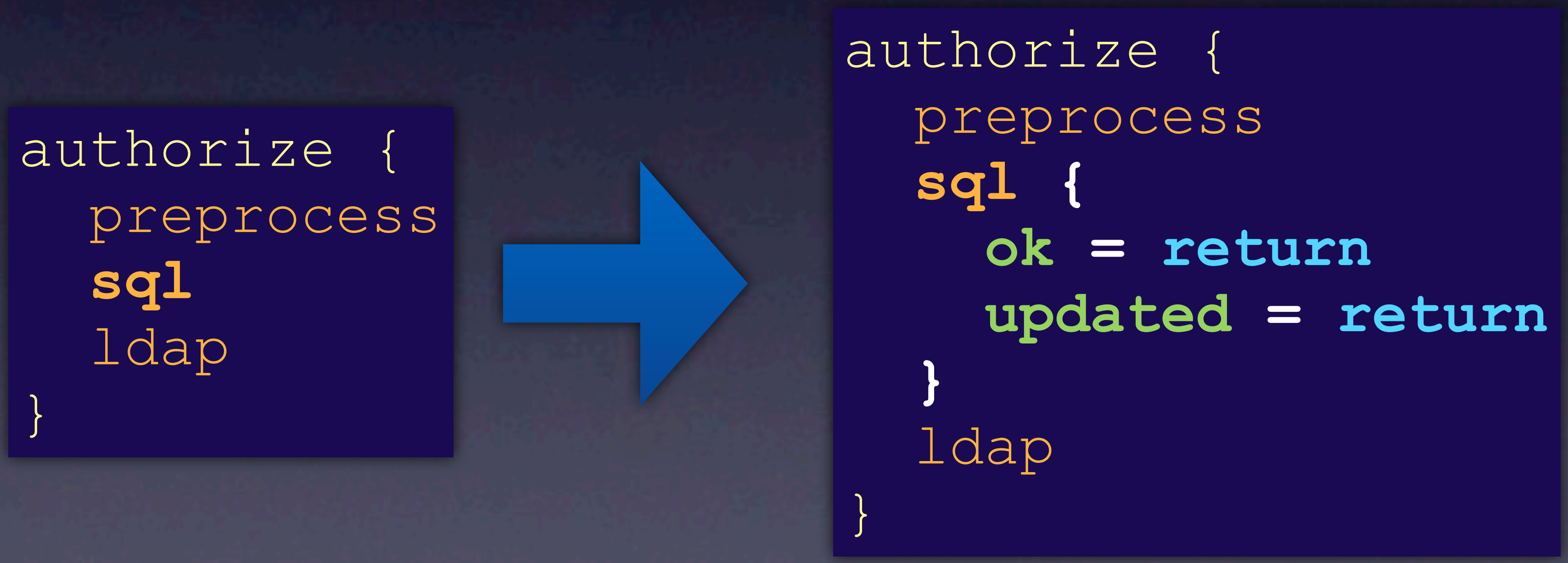
Return codes priorities

The following table indicates the default level of priority for each possible return code. The `return` priority indicates that the section will immediately stop if the corresponding code is returned.

Code	Meaning	Priority
`notfound`	User was not found	1
`noop`	Module is not applicable (it did nothing)	2
`ok`	User accepted	3
`updated`	User accepted and attribute list updated	4
`fail`	Module failed (ex: database access failure)	**return**
`reject`	User rejected	**return**
`userlock`	User rejected because his account is locked	**return**
`invalid`	User rejected because his configuration is invalid	**return**
`handled`	Module handled the request and its response (if any)	**return**

Modifying priorities

- In some cases, the default priorities need to be modified
- For example, one might want to stop immediately if a module answers **ok**
- To do this, simply append a section to the module name, and set the desired priorities in that section, for example:



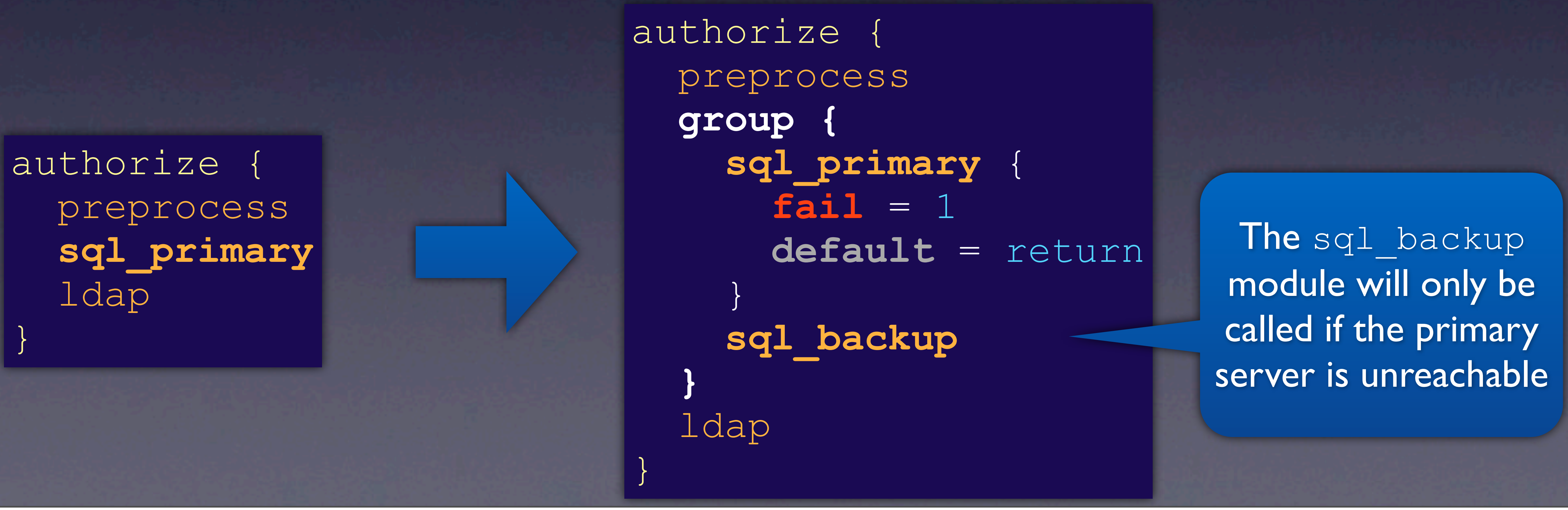
```
authorize {
  preprocess
  sql
  ldap
}
```

```
authorize {
  preprocess
  sql {
    ok = return
    updated = return
  }
  ldap
}
```

- In this example, if the `sql` module returns **ok** (or **updated**), then the `authorize` section will stop immediately and will itself return **ok** (or **updated**): the `ldap` module will not be called

Grouping modules

- Multiple modules may be grouped in a `group` section
- Modules in a group are called one after the other, each one returning a code, and the return code with the highest priority is returned by the group itself (the group handling process is interrupted if a module's code has a **`return`** priority level)
- This can be useful to implement a *fail-over* mechanism between modules, for example:



```
authorize {
  preprocess
  sql_primary
  ldap
}
```

```
authorize {
  preprocess
  group {
    sql_primary {
      fail = 1
      default = return
    }
    sql_backup
  }
  ldap
}
```

The `sql_backup` module will only be called if the primary server is unreachable

Group priorities

- You can also modify the priority rules for the return code of the group itself
- For example, imagine you don't want to query the LDAP server if either the primary or the secondary SQL server has answered **ok** (or **updated**):

```
authorize {
 preprocess
 sql_primary {
  ok = return
  updated = return
 }
 ldap
}
```

→

```
authorize {
 preprocess
 group {
  sql_primary {
   fail = 1
   default = return
  }
  sql_backup
  ok = return
  updated = return
 }
 ldap
}
```

`redundant` sections

- For *fail-over*, it is generally simpler to use a `redundant` section instead of a `group` section
- It's the same thing, except that the default priority rules in a `redundant` section are **`fail`** = **`1`** and **`default`** = **`return`**

```
authorize {
 preprocess
 group {
  sql_primary {
   fail = 1
   default = return
  }
  sql_backup1 {
   fail = 1
   default = return
  }
  sql_backup2

  ok = return
  updated = return
 }
 ldap
}
```

→

```
authorize {
 preprocess
 redundant {
  sql_primary
  sql_backup1
  sql_backup2

  fail = return
  ok = return
  updated = return
 }
 ldap
}
```

Load balancing

- To load-balance requests between multiple modules (for example to hit three different database servers), simply use a `load-balance` section:

```
authorize {
 preprocess
 load-balance {
  sql1
  sql2
  sql3
 }
}
```

- One of the modules is chosen randomly and executed, and its result is returned by the `load-balance` section itself (even if the module returns **`fail`**)
- If you want to fallback to one of the remaining modules in case a module returns **`fail`**, then you should use a `redundant-load-balance` section: the section only fails if *all* its modules fail

The `unlang` «langage»

- Before version 2 of freeRADIUS, if you wanted to express a condition in the configuration of the request handling policy, you generally had no other option than to write your own module
- Now, if you need to express relatively simple conditions, you may do so using `if`, `else`, `elsif`, etc., for example:

```
authorize {
 preprocess
 if (User-Name == "joe") {
  ldap1
 }
 elsif (User-Name == "jack") {
  ldap2
 }
 else {
  sql
 }
}
```

In this example, if the request concerns user `joe`, then use the `ldap1` module, or else if it's `jack`, then use module `ldap2`, or else use the `sql` module (for all other users).

Note: a `load-balance` or `redundant-load-balance` section must not contain any `else` or `elseif` subsection. A `redundant` section must not contain `if`, `else` or `elseif` sections at all.

The `unlang` «langage»

- The `unlang` language is not a full-featured language, and does aim at becoming one (hence its name): its only goal is to express simple rules (if you need some complex logic, you need to write a module, see later)

- The condition of an `if` section can be:
  - `(attribute operator value)`
  - `(value)`
  - `(return_code)`

Example: `(Session-Timeout >= 3600)`

True if the value is not null or empty, for example: `(Idle-Timeout)` or `("%{Idle-Timeout}")`

True if the last called module returned this code, ex: `(fail)`

- Just like in the C language, you may use `!a` to express «not a», `a && b` for «a and b», and `a || b` for «a or b»

- You may nest conditions, for example: `(a && !(b || c))`

The unlang «langage»

- You may also use the switch / case statement, very much like in the C language:

```
authorize {
 preprocess
 switch "%{User-Name}" {
  case "joe" {
    ldap1
  }
  case "jack" {
    ldap2
  }
  case {
    sql
  }
 }
}
```

This string may contain xlats...

...but the values in case statements may not

This is the default (fallback) section

- This example will have the same result as the one we saw earlier with the if, elsif and else instructions

The `unlang` «langage»

- By default, attributes are looked up in the `request` list
- You may specify another internal list using the following syntax: `%{list:attribute}` for example `%{control:Auth-Type}`
- So far we have talked about the `request`, `control` and `reply` lists, but there are a few other lists:
  - `proxy-request` and `proxy-reply` contain the attributes that are sent to or received from a *Home-Server*, when freeRADIUS acts as a *Proxy-Server*
  - `outer.request`, `outer.reply`, `outer.control`, `outer.proxy-request`, and `outer.proxy-reply` allow you to access the attribute lists of the outer EAP request during the handling of the inner EAP request of a PEAP or TTLS tunnel

The `unlang` «langage»

- A few other xlat options exist:
  - `%{#string}`: length of the string
  - `%{attribute[n]}`: (n+1)th attribute of this type
  - `%{attribute[#]}`: number of attributes of this type
  - `%{attribute[*]}`: all values of attributes of this type, separated by line feeds (`\n`) and grouped into one string
  - `%{0}`: the string identified by the last regular expression (with `=~` or `!~`)
  - `%{1}`, `%{2}`, ..., `%{8}`: the groups identified by the last regular expression
- However, should you need this level of complexity, you probably should consider writing a module instead (in python, perl or C, as we will se later)

The `unlang` «langage»

- You may also define an `update` section, which allows you to update an attribute list very simply, for example:

```
update reply {
  Reply-Message := "Bonjour %{User-Name}"
  Session-Timeout <= 3600
  Filter-Id !* ALL
}
```

- All the lists can be modified (`request`, `control`...)
- The `=`, `:=` and `+=` operators have the same meaning as described earlier, and you may also use the following operators:

`-=` : deletes all attributes of this type with the given value
`==` : deletes all the attributes of this type, except those with the given value
`!*` : deletes all attributes of this type (whatever the given value)
`<=` : replaces the values greater than the given value with that value (integer attributes only)
`>=` : replaces the values lower than the given value with that value (integer attributes only)

For the `<=` and `>=` operators, the attribute is added with the given value if it does not exist

always module

- The always module always returns the same code, which is configurable. Here’s the default configuration for this module:

```
always fail {
     rcode = fail
}
always reject {
     rcode = reject
}
always noop {
     rcode = noop
}
always handled {
     rcode = handled
}
always updated {
     rcode = updated
}
always notfound {
     rcode = notfound
}
always ok {
     rcode = ok
     simulcount = 0
     mpp = no
}
```

Here’s an example that uses the reject variant of the always module

```
authorize {
  preprocess
  if (User-Name=="bad-guy") {
    reject
  }
  ...
}
```

policy.conf

- If the same piece of *unlang* code needs to be used in several places, it is best to define a section containing that code in the `policy` section located in `/etc/freeradius/policy.conf`, for example:

```
policy {
  add_welcome_message {
    update reply {
      Reply-Message := "Hello %{User-Name}"
    }
  }
}
```

- This «function» can then be used elsewhere in the configuration:

```
authorize {
  preprocess
  add_welcome_message
  ...
}
```

`policy` module

- Another module, named `policy`, offers a similar functionality, although somewhat more limited
- **WARNING**: this module has nothing to do with the `policy.conf` file that we have just seen
- It is now preferrable to use `policy.conf`, and simply ignore the `policy` module
- Note: the `policy` module relies on a configuration file which can also be ignored: `/etc/freeradius/policy.txt`

Default policy

- The default request handling policy is defined in `/etc/freeradius/sites-enabled/default`
- It's just a symbolic link to the file `/etc/freeradius/sites-available/default`
- This file includes the previously described sections: `authorize` and `authenticate`, as well as other similar sections...

Other modules sections

There are a few other sections that call modules in much the same way as the **`authorize`** and **`authenticate`** sections:

- **`session`**: if the `Simultaneous-Use` attribute is added to the control list (for example by the `files` module during the `authorize` phase), then the modules listed in the `session` section will make sure that the maximum number of sessions currently opened by the user is lower than the value of this attribute, and reject the user if the number is reached
- **`post-auth`**: actions to be executed after authentication

Other modules sections

- If a request is rejected at any moment during the `authorize` or `authenticate` phases, then the '**`Post-Auth-Type REJECT`**' subsection of the **`post-auth`** section will be run
- This is often used to add attributes in the `Access-Reject` response
  - for example to add a `Reply-Message` attribute containing an error message that the NAS can then display to the user

Other modules sections

Two modules sections are executed when hanlding an `Accounting-Request` :

- `preacct`: list of modules executed before accounting
- `accounting` : list of modules that handle the accounting itself

Other modules sections

And finally two modules sections are executed by the freeRADIUS server, before and after a packet is proxied to a Home-Server, in a roaming context:

- `pre-proxy`: modules executed before proxying a packet
- `post-proxy`: modules executed when the response is received from the Home-Server

Virtual servers

All the policy sections that we have seen may also be defined inside a named server section ‘**server** ***virtual_server_name***’:

```
server ldap-policy {
 authorize {
  preprocess
  ldap
 }
 authenticate {
  Auth-Type LDAP {
   ldap
  }
 }
 ...
}
```

This is called a «virtual server»

Virtual servers

- Multiple virtual servers may be defined, each one with its own request handling policy
- Each virtual server is usually configured in its own file in the `sites-available` directory...
- ...and symbolic links must be created in the `sites-enabled` pointing to the virtual server files that you want to enable

Virtual servers

Once the virtual servers are defined, freeRADIUS may be configured to dynamically select the appropriate virtual server for each request (in other words, it may select dynamically which policy must be applied), depending on:

- the IP address and UDP port where the packet was received: in the corresponding `listen` section, simply add `virtual-server=virtual-server-name`
- the NAS that sent the request, by adding the same statement in the appropriate `client` section
- the Home-Server to which the packet is proxied, in case of roaming, again with the same statement in the Home Server's configuration (may be useful to define `pre-proxy` and `post-proxy` sections specific to each roaming partner)

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
  - Configuration of the RADIUS dictionary
  - Basic configuration of the server
  - Request management policies configuration
  - Modules configuration
  - Roaming configuration

Modules configuration

- Apart from a few exceptions (`eap.conf`, `sql.conf`...), the configuration of all modules is located in the files of the `/etc/freeradius/modules` directory
- The configuration has the following format:

```
module_name {
 a_param = 23
 another_param = "blabla"
 ...
}
```

OR

```
module_name another_name {
 a_param = 23
 another_param = "blabla"
 ...
}
```

- If you specify another_name, it is this name that must be used in the rest of the configuration

Modules configuration

- For example, here’s the **`files`** module’s config:

```
files {
    # The default key attribute to use for matches.  The content
    # of this attribute is used to match the "name" of the
    # entry.
    #key = "%{Stripped-User-Name:-%{User-Name}}"

    usersfile = ${confdir}/users
    acctusersfile = ${confdir}/acct_users
    preproxy_usersfile = ${confdir}/preproxy_users

    #  If you want to use the old Cistron 'users' file
    #  with FreeRADIUS, you should change the next line
    #  to 'compat = cistron'.  You can the copy your 'users'
    #  file from Cistron.
    compat = no
}
```

`/etc/freeradius/modules/files`

Modules configuration

- Another example, here's the `realm` module's config:

```
realm suffix {
    format = suffix
    delimiter = "@"
}

realm realmpercent {
    format = suffix
    delimiter = "%"
}
```

/etc/freeradius/modules/realm

- You may use those two variants of the `realm` module in the rest of the configuration, by using the names `suffix` and `realmpercent`

Modules configuration

- The configuration of some modules may be organized in subsections in order to group related parameters, for readability
- For example, the subsection `tls` in the `ldap` module:

```
ldap {
    server = "ldap.example.com"
    identity = "cn=admin,dc=example,dc=com"
    ...
    tls {
        start_tls = no
        cacertfile = /path/to/cacert.pem
        ...
    }
    ...
}
```

`/etc/freeradius/modules/ldap`

EAP configuration

- EAP configuration is also organized in subsections:

```
eap {
    default_eap_type = md5
    timer_expire     = 60
    ...
    md5 {
    }
    ...
    tls {
        certdir = ${confdir}/certs
        cadir = ${confdir}/certs
        ...
    }
    ...
    peap {
        default_eap_type = mschapv2
        copy_request_to_tunnel = yes
        use_tunneled_reply = no
    #   proxy_tunneled_request_as_eap = yes
        virtual_server = "inner-tunnel"
    }
    mschapv2 {
    }
}
```

/etc/freeradius/eap.conf

Some submodules (such as md5) have no configuration, but you need to add a section if you want to enable them

TLS configuration is required for EAP/TLS, PEAP and TTLS

If this parameter is defined, then the inner EAP requests will be handled by the given virtual server, or else it will be the same virtual server that handled the external EAP dialog

This is one of the rare modules whose configuration is *not* located in the `modules` directory

Modules instantiation

- When freeRADIUS starts up, it parses the configuration files and determines the list of all the modules that can possibly be used
    - ➡ Oddly enough, it ignores the modules that are used in the translations (for example: `%{expr:2+3}`)
- It creates an instance of each of those modules, and calls their initialization function
- If you want to specify the order of the instantiation, or to load extra modules (such as the ones used only in translations), simply list those modules in the `instantiate` section of `radiusd.conf`:

```
instantiate {
  exec
  expr
  sql
}
```

Modules instantiation

Here's an example where using the `instantiate` section is compulsory:

- The `sql` module may be configured to load the list of NASes from the `nas` database table (this list is added to the list loaded from `clients.conf`)
- This happens during the module's instantiation
- If you want to use the database only to manage the NAS list, then you <u>must</u> add the `sql` module to the `instantiate` section, since it will not be used elsewhere

Virtual modules

- If you define a named section in the `instantiate` section, then it is considered as the definition of a «virtual module», for example:

```
instantiate {
  ...
  redundant redundant_sql {
    sql1
    sql2
    sql3
  }
}
```

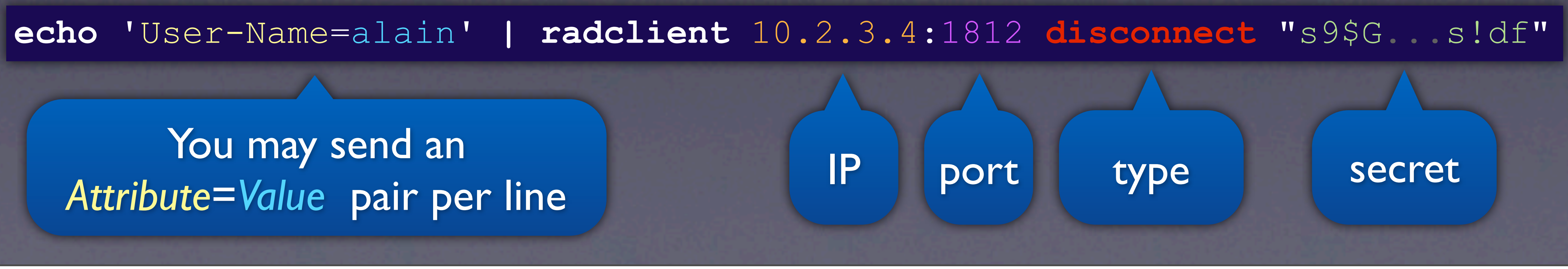
- You may use a virtual module anywhere in the configuration, just like a regular module:

```
authorize {
  preprocess
  redundant_sql
}
```

- You can achieve the same result using `policy.conf` instead, as we have seen earlier

Change of Authorization (CoA)

- The RADIUS protocol did not initially define any mechanism to allow you to ask a NAS to disconnect a user, or to ask a NAS to change a connected user's access rights
- In RFC 3576, two new types of RADIUS requests were defined for this: type `disconnect` to disconnect a user, and type `coa` to change a user's authorizations (when we speak of CoA in a general sense, we mean both types of requests)
- Warning: with CoA, the NAS is actually acting as a server, and anyone can act as a client, as long as he shares a secret with the NAS
- For example, here's how to send a request to the NAS at IP address `10.2.3.4`, port `1812`, to disconnect user `alain`:



```
echo 'User-Name=alain' | radclient 10.2.3.4:1812 disconnect "s9$G...s!df"
```

You may send an *Attribute*=*Value* pair per line

IP

port

type

secret

CoA from freeRADIUS

- Although the CoA requests may be sent by anyone, it is sometimes useful to have the freeRADIUS server send them, for example:
  - If you want to disconnect a user from one NAS when he connects to another NAS
  - If you want to have freeRADIUS send a `coa` request to the user's NAS if it notices that the user's rights have changed (when handling an `Interim-Update`, for example)
  - If you want to be able to send a request to freeRADIUS so that it finds the NAS that a user is connected to and sends a CoA request to that NAS
- *Note*: freeRADIUS cannot (yet) receive CoA requests, and cannot proxy CoA requests from a *Home-Server* to a NAS

CoA from freeRADIUS

- CoA is not supported by many NASes
- Its configuration in freeRADIUS is still young and may change in future versions
- For more info, read:

```
/etc/freeradius/sites-available/originate-coa
```

Organization

- The configuration lives in files located in `/etc/freeradius` and its subdirectories (on other systems than Debian, it lives in `/etc/raddb`)
- For this presentation, we will cut the configuration in five parts:
  - Configuration of the RADIUS dictionary
  - Basic configuration of the server
  - Request management policy configuration
  - Module configuration
  - Roaming configuration

Roaming example

- Reminder: when acting as a proxy-server in a roaming scenario, freeRADIUS proxies some requests to one or more Home-Servers
- To know which requests must be proxied, and which Home-Server they must be proxied to, the preferred solution is usually to base the decision on the `User-Name`
- For example, you can configure freeRADIUS to make it proxy requests whose `User-Name` is `joe%foo.com` to the RADIUS server at `rad1.foo-telecom.net`.
- In our example, we will proxy requests to a secondary server if the primary server is down

Identifying the *realm*

- The first step is to identify requests that must be proxied to a *Home-Server*
- To do this, freeRADIUS looks for the **`Realm`** attribute in the **`control`** list, after the authorization phase
- If this attribute exists, then the packet is proxied to the Home-Server (or one of the Home-Servers) configured for the given realm
- The **`realm`** module is generally used to set the **`Realm`** attribute during the authorization phase, based on a prefix or suffix found in the `User-Name` attribute

The `realm` module

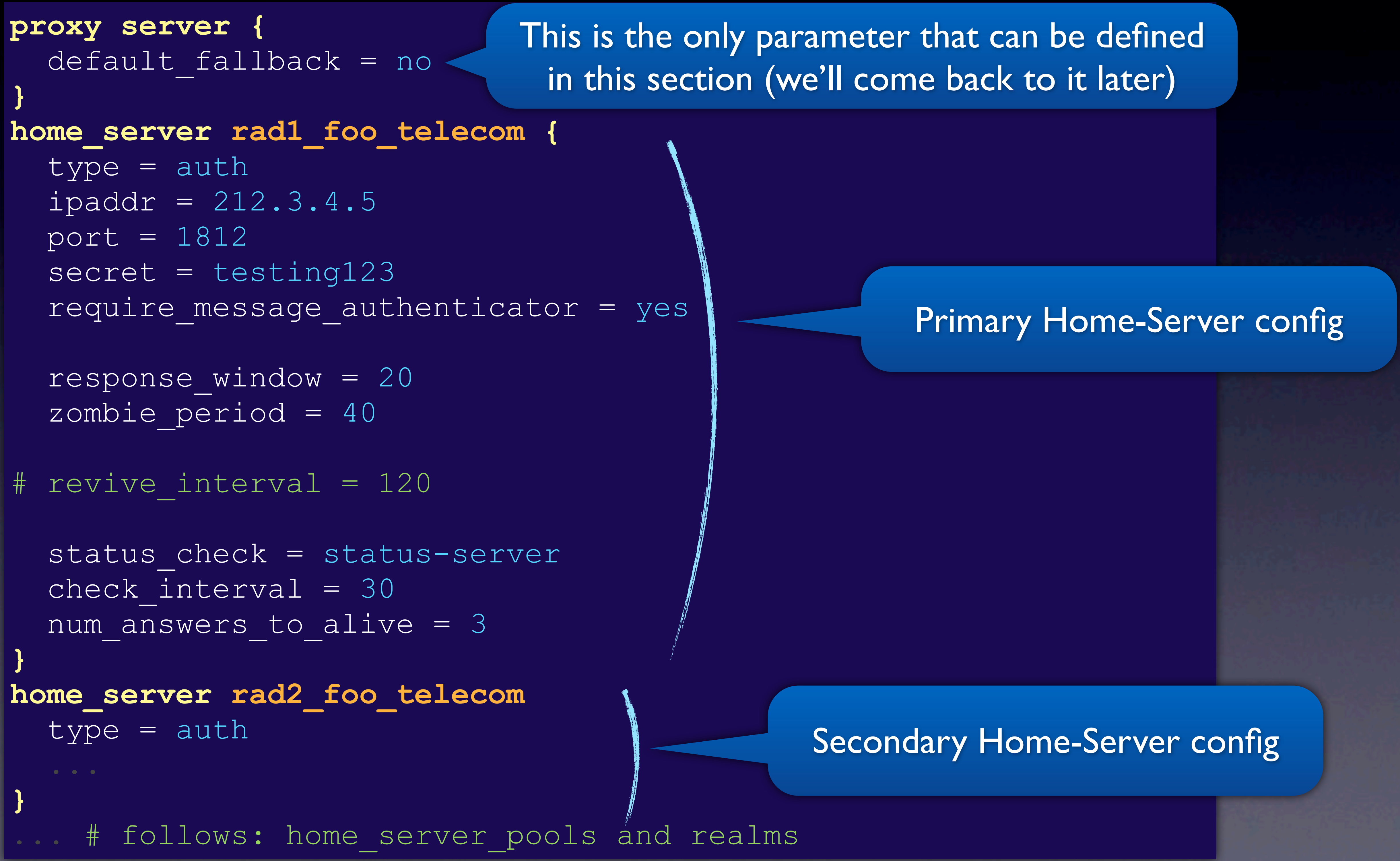
- As we have seen earlier, the default configuration of the `realm` module defines the `realmpercent` variant
- In our example, we just need to add the `realmpercent` module to the `authorize` section, and the user's realm will be identified as the part of the `User-Name` after the `%` character
- When the server receives a request from **`joe%foo.com`**, the `realmpercent` module will add the **`Realm`** attribute in the **`control`** list, set to «**`foo.com`**», during the authorization phase
- This module will also add the **`Stripped-User-Name`** attribute in the **`control`** list, set to «**`joe`**»

proxy.conf

- The heart of the roaming configuration is located in `proxy.conf`
- This file is composed of multiple sections:
  - a **`proxy server`** section for general roaming settings
  - a **`home_server`** section for each Home-Server
  - one or more **`home_server_pool`** sections that allow you to define rules to load-balance requests between several Home-Servers
  - One or many **`realm`** sections that indicate which **`home_server_pool`** must be used for each realm

proxy.conf

- Here’s an example of `proxy.conf` configuration:



```
proxy server {
   default_fallback = no
}
home_server rad1_foo_telecom {
   type = auth
   ipaddr = 212.3.4.5
   port = 1812
   secret = testing123
   require_message_authenticator = yes

   response_window = 20
   zombie_period = 40

#  revive_interval = 120

   status_check = status-server
   check_interval = 30
   num_answers_to_alive = 3
}
home_server rad2_foo_telecom
   type = auth
   ...
}
... # follows: home_server_pools and realms
```

This is the only parameter that can be defined in this section (we’ll come back to it later)

Primary Home-Server config

Secondary Home-Server config

proxy.conf

```
}
home_server rad1_foo_telecom {
  type = auth
  ipaddr = 212.3.4.5
  port = 1812
  secret = testing123
  require_message_authenticator = yes

  response_window = 20
  zombie_period = 40

# revive_interval = 120

  status_check = status-server
  check_interval = 30
  num_answers_to_alive = 3
}
home_server rad2_foo_telecom
  type = auth
  ...
}
... # follows: home_server_pools and realms
```

auth or acct or auth+acct

The IP address and UDP port where the requests must be proxied, and the secret shared with this Home-Server (as far as the Home-Server is concerned, the proxy-server is just like a regular NAS)

It is preferable to enter an IP address rather than a host name, because if the DNS request fails then freeRADIUS cannot start up

Does this Home-Server expect a **Message-Authenticator** attribute in each request? If so, freeRADIUS adds it.

These settings can prevent freeRADIUS from proxying requests to a dead Home-Server

proxy.conf

```
proxy server {
  default_fallback = no
}
home_server rad1_foo_telecom {
  type = auth
  ipaddr = 212.3.4.5
  port = 1812
  secret = testing123
  require_message_authentic

  response_window = 20
  zombie_period = 40

# revive_interval = 120

  status_check = status-server
  check_interval = 30
  num_answers_to_alive = 3
}
home
  ty
  . .
}
. . .
```

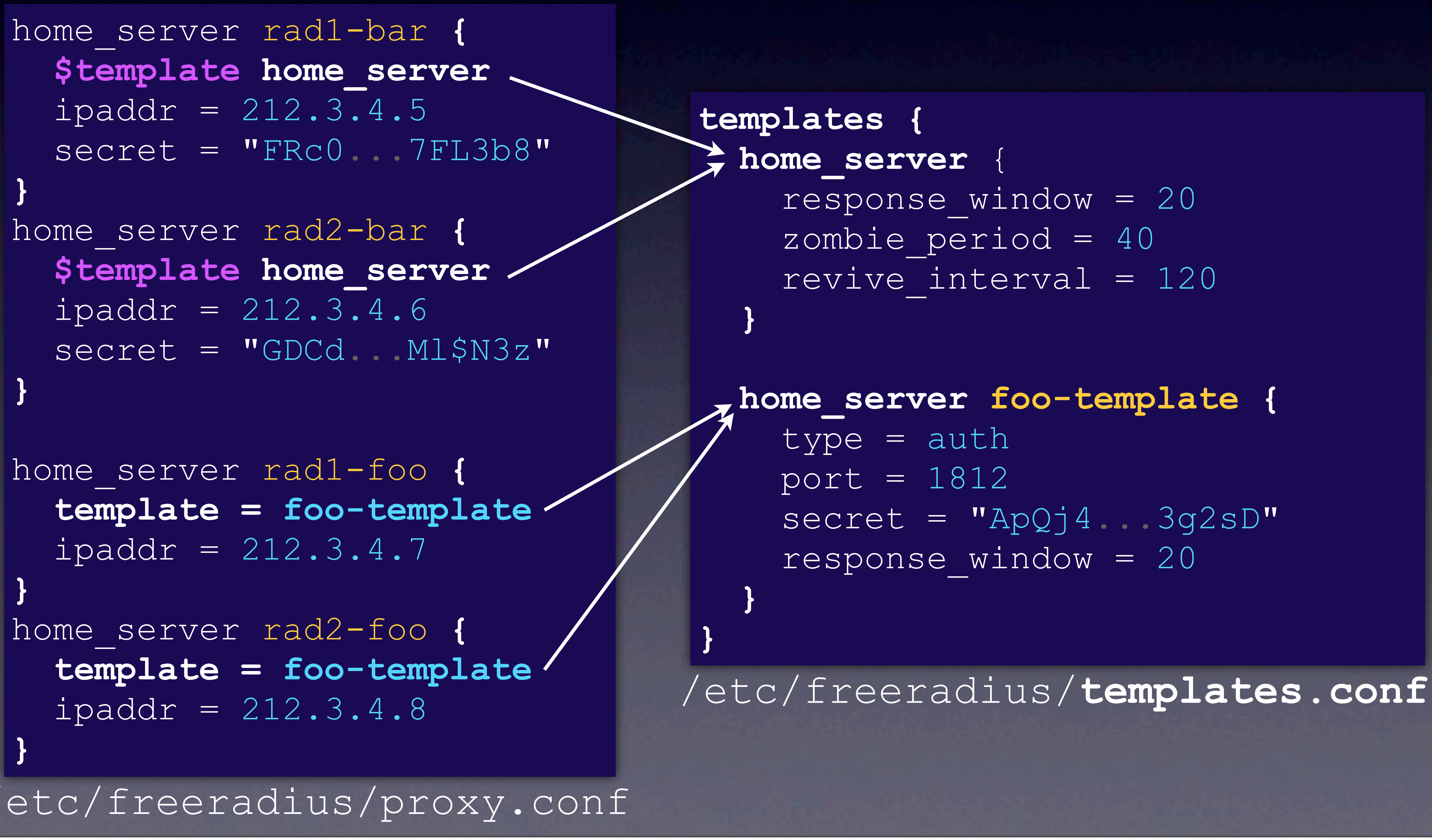
If this Home-Server does not respond during 20s, then it is considered a **zombie** (it will only be queried if no Home-Server is **alive**). After 40s, the Home-Server is considered really **dead** (never queried). If you set `revive_interval=120`, then it will be considered **alive** again after 2 minutes (even if it is not)...

...but it is preferable to send `status` requests to the Home-Server at regular intervals instead: in this example, we query the Home-Server every 30s and it takes 3 successive successes to revive it

If we use `status` requests, then the Home-Server must be configured to handle them, of course. If it's a freeRADIUS server, you must create a `listen` section in its config with the `status` type, then set `status_server=yes` in the `security` section, and finally create a virtual server to handle `status` requests (see `sites-available/status`)

templates.conf

- The Home-Servers' settings are often very similar
- To avoid repetitions, you may define configuration templates:



```
home_server rad1-bar {
  $template home_server
  ipaddr = 212.3.4.5
  secret = "FRc0...7FL3b8"
}
home_server rad2-bar {
  $template home_server
  ipaddr = 212.3.4.6
  secret = "GDCd...Ml$N3z"
}

home_server rad1-foo {
  template = foo-template
  ipaddr = 212.3.4.7
}
home_server rad2-foo {
  template = foo-template
  ipaddr = 212.3.4.8
}
```

/etc/freeradius/proxy.conf

```
templates {
  home_server {
    response_window = 20
    zombie_period = 40
    revive_interval = 120
  }

  home_server foo-template {
    type = auth
    port = 1812
    secret = "ApQj4...3g2sD"
    response_window = 20
  }
}
```

/etc/freeradius/**templates.conf**

templates.conf

- Templates can be used in any section in the configuration, exception subsections (only root sections can use templates)
- This can be useful, for example, for the definition of the NASes in `clients.conf`
- For sub-sections, you can achieve something quite similar using the **`$INCLUDE`** instruction

proxy.conf

- Let’s now focus on the rest of the proxy.conf file:

```
...
home_server_pool foo_telecom_pool {
  type = fail-over
  virtual_server = pre_post_proxy_for_foo
  home_server = rad1_foo_telecom
  home_server = rad2_foo_telecom
}
realm foo.com {
  auth_pool = foo_telecom_pool
  nostrip
}
```

In this example, we configure a pool composed of the two Home-Servers defined earlier

All the Home-Servers in a pool must be have the same type (auth or acct or auth+acct)

And finally, we point the foo.com realm to this pool

- either use auth_pool (for Home-Servers of type auth) and/or acct_pool (for Home-Servers of type acct)
- or use pool (for Home-Servers of type auth+acct)
- or finally use no pool at all, in which case the realm is handled locally (no proxying to Home-Servers)

proxy.conf

This pool’s type is `fail-over`, meaning that the request is proxied to the first Home-Server, and if it does not answer, then the secondary server is called, and so on

```
...
home_server_pool foo_telecom_pool {
  type = fail-over
  virtual_server = pre_post_proxy_for_foo
  home_server = rad1_foo_telecom
  home_server = rad2_foo_telecom
}
realm foo.com {
  auth_pool = foo_telecom_pool
  nostrip
}
```

A virtual server may be set, in which case its `pre-proxy` and `post-proxy` sections will be executed before the request is proxied, and after the response is received from the Home-Server

By default, if a `Stripped-User-Name` attribute is present in the `control` list, then its value is used for the `User-Name` attribute in the request that is proxied to the Home-Server. For example, the Home-Server will receive a request for `joe`, and not for `joe%foo.com`. The `nostrip` option allows you to specify that you want to keep the original `User-Name` (this can be useful if the Home-Server also acts as a proxy server for example).

Other types of pools

We have just seen the `fail-over` pool type, but other types exist:

- `load-balance`: each request is randomly sent to one of the Home-Servers (with a preference for the Home-Servers that respond well)
    - ***Warning***: the EAP authentication methods will probably not work with this pool type, because they require multiple successive requests to the same server
- `client-balance`: also random, but all the requests from a given NAS are always proxied to the same Home-Server (as long as it is **alive**)
- `keyed-balance`: again random, but all the requests that have the same `Load-Balance-Key` attribute value will be proxied to the same Home-Server
    - ➡ A module must therefore add this attribute to the `control` list, for example by copying the value of the `User-Name` attribute (so that all the requests from a given user will be proxied to the same Home-Server)

NULL and LOCAL realms

- If you define a realm named **NULL**, then it is used for all requests that do not have a realm

- Many people define a real called **LOCAL** with no pool (it will therefore be handled locally): you can then force a requests to be handled locally by adding the attribute Proxy-To-Realm in the **control** list, with its value set to «**LOCAL**»

- For example, you generally do not want to proxy the content of a PEAP or TTLS tunnel to another server (for security reasons). To ensure this, you can add the following 3 lines to the configuration of the inner-tunnel virtual-server:

```
update control {
     Proxy-To-Realm := LOCAL
}
```

Virtual Home-Server

- If you define no settings in a `home_server` section except for the `virtual-server` setting, then all requests proxied to this «virtual Home-Server» will be handled locally by the chosen virtual-server
- For example:

```
home_server virtual_home_server_for_foo {
  virtual_server = virtual_server_for_foo
}
```

- This is useful for example to execute some code when all Home-Servers of a pool have failed:

```
home_server_pool foo_telecom_pool {
  type = fail-over
  home_server = rad1_foo_telecom
  home_server = rad2_foo_telecom
  home_server = virtual_home_server_for_foo
}
```

This is a pool of type `fail-over`, so `rad1` is tried first, and if it fails, then it tries `rad2`, and if it fails too, then the virtual server is called

Fallback Home-Servers

- In the configuration of a **home_server_pool** you can define a fallback Home-Server, that will be used if all Home-Servers in the pool are dead (the fallback server is often a virtual Home-Server) :

```
home_server_pool foo_telecom_pool {
  type = load-balance
  home_server = rad1_foo_telecom
  home_server = rad2_foo_telecom
  fallback = virtual_home_server_for_foo
}
```

In this example, the load is balanced between servers `rad1` and `rad2`. If both servers die, then the pool falls back to **`virtual_home_server_for_foo`**.

- If no fallback server is defined, and if the `default_fallback` option is set to `yes` in the **proxy server** section, then the **DEFAULT** realm is used when all the Home-Servers of a realm are dead

➡ *The* **DEFAULT** *realm is often configured with a simple pool containing a single virtual Home-Server pointing to a virtual server that logs the failure.*

Filtering attributes

- In a roaming context, it is often necessary to make sure that the attributes returned by the Home-Server are acceptable, and remove them if they are not

- This is the role of the **attr_filter** module. Here's an extract of its default configuration:

```
attr_filter attr_filter.post-proxy {
    attrsfile = ${confdir}/attrs
}
...
```

/etc/freeradius/modules/attr_filter

- You can add **attr_filter.post-proxy** in the **post-proxy** section: when freeRADIUS will receive a response from a Home-Server, it will apply the filtering rules defined in: /etc/freeradius/attrs

- This file specifies, for each realm, which attributes are acceptable, and with what values: non-compliant attributes are removed

The `attrs` file

- The `attrs` file is composed of a list of rules, somewhat like the `users` file, but with a different rational, for example:

```
foo.com
        Reply-Message =* ANY,
        Session-Timeout <= 86400,
        Idle-Timeout <= 600,
        Acct-Interim-Interval >= 300,
        Acct-Interim-Interval <= 3600

...
```

`/etc/freeradius/attrs`

Tab, not spaces

The condition operators are identical to those used in the `users` file, plus the `:=` operator (which adds the attribute or replaces it if it does not already exist)

- A rule starts with the name of a realm, then a list of conditions to apply on attributes, each on one line
- The **`attr_filter`** module starts by looking for a rule that matches the packet's **`Realm`**, then it deletes all the attributes that are not listed in the rule, and also deletes all the attributes that do not satisfy any of the listed conditions

The `attrs` file

- You may define a **`DEFAULT`** rule (only one): it is used if no realm matches
- You may also add **`Fall-Through = Yes`** at the end of a rule to specify that you also the want the conditions of the **`DEFAULT`** rule to be applied:

```
low-budget-telecom
    Filter-Id := "limited-service",
    Fall-Through = Yes

DEFAULT
    Login-TCP-Port <= 65536,
    Framed-MTU >= 576,
    Filter-ID =* ANY,
    Reply-Message =* ANY,
    Proxy-State =* ANY,
    EAP-Message =* ANY,
    Service-Type == Framed-User,
    Service-Type == Login-User,
    ...
    Message-Authenticator =* ANY,
    State =* ANY,
    Session-Timeout <= 28800,
    Idle-Timeout <= 600,
    Port-Limit <= 2
```

`/etc/freeradius/attrs`

In this example, all the responses from the Home-Servers of the **`low-budget-telecom`** realm will have the attribute **`Filter-Id`** set to **`limited-service`** (if this attribute already exists, then its value is replaced), then the **`DEFAULT`** filtering is applied

You may specify multiple authorized values for an attribute: an attribute is kept if it matches any condition

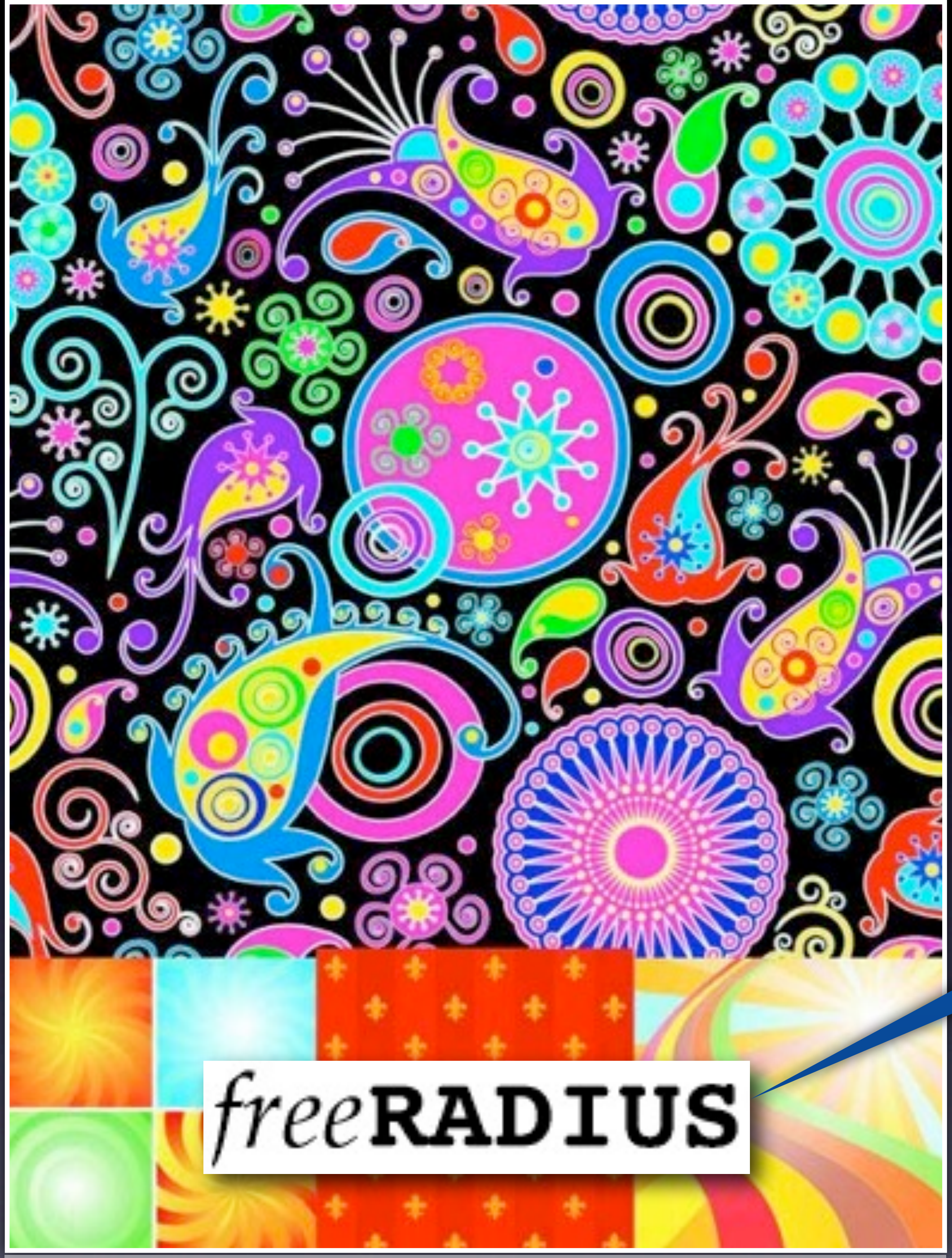
Filtering attributes

The `attr_filter` module may also be used to filter out attributes in other contexts, using the same principles:

- before a request is proxied to a Home-Server (see the `attrs.pre-proxy` file)
- or even outside the context of roaming: for example, attributes may be filtered out depending on the user (rather than on the realm) by setting `key = %{User-Name}` in the `attr_filter` module's configuration
    - ➡ in the rules definition file, instead of specifying the name of a realm at the beginning of a rule, you would then specify the name of a user

Wow! You know everything about freeRADIUS configuration files!

In the rest of this presentation, we will detail a few more useful modules and see how to create new modules



Questions?