Chapter 5 - Basic Authentication Methods

The following topics are discussed in this chapter:

- Password Authentication Protocol (PAP)
- Password formats
- Alternate authentication methods
- Forcing Authentication

5.0 Password Authentication Protocol (PAP)

Configuring the Password Authentication Protocol (PAP) is the first step in FreeRADIUS authentication. This protocol compares a password entered by the user to the "known good" password contained in the authentication system. Although other authentication protocols may be used, PAP is the simplest and easiest of these to configure. Authentication testing should, therefore, always be started with PAP, because once that works, it will be easier to configure the other authentication protocols.

5.0.1 Correct User and Password

This section demonstrates how to configure PAP via the `users` file. The `users` file is a flat-text file that allows many simple policies to be implemented. It is simple to use, easy to edit, and does not require additional configuration (such as with an LDAP or SQL). This file is therefore the ideal configuration file to use when deploying a new server.

The following authentication types will work with the default configuration in Version2.x and the above `users` file entry:

- PAP
- CHAP
- MS-CHAP
- EAP-MD5
- EAP-MSCHAPv2
- Cisco LEAP

If the Validate Server Certificate checks on the 802.1x supplicant is disabled strictly for testing, the following authentication types will also work:

- PEAPv0
- EAP-GTC
- EAP-MSCHAPv2
- EAP-TTLS

As indicated by the long list of authentication options, FreeRADIUS is flexible and supports authentication of users by a variety of methods. The basic requirement is to configure a user with a password.

The Test

In the example below, PAP authentication is configured by instructing the server to identify a particular user ("bob") and the user's "known good" password ("hello"). The user's "known good" password, listed in

the `users` file, is validated against the password sent to the server by the client, as entered by the user. If the passwords match, then the server will return an `Access-Accept` packet. If the passwords do not match, then the server will return an `Access-Reject` packet.

1. The following text, placed at the first line of the `users` file, tells the server about the user and the "known good" password. If the text is not placed in the correct position at the first line of the `users` file, the test will not work. When entering the name ("bob"), make sure the first letter ("b") is the first character on the line. Make sure there are no spaces or other characters preceding the text.
   ```
   bob Cleartext-Password := "hello"
   ```
2. Start the server by running the following command from the shell:
   ```
   $ radiusd -X
   ```
3. Wait for the text to stop scrolling by.
   The final printed out line should be: `Ready to process requests.`
4. In another terminal window on the same machine, type the following command:
   $ radtest -x bob hello 127.0.0.1 0 testing123
   This command sends an authentication request packet to the server, with the following parameters:
   - Print more debugging information -x
   - Name bob
   - Password hello
   - To server 127.0.0.1
   - Pretending to log in on switch port 0
   - using a shared secret of testing123

If the authentication process is successful, the server returns an `Access-Accept` message, and the radtest window shows the following message::

```
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=218,
length=20
```

The Server Output

Switch back to the terminal window where the server is running - a lot of additional output should be visible. This output contains information about what the server does as it processes the authentication request. The final lines should be:

```
Sending Access-Accept of id 218 to 127.0.0.1 port 62977
Finished request 0.
Going to the next request
Waking up in 4.9 seconds.
```

This message indicates that the server has sent an `Access-Accept` to the client and has marked the request as finished. At this point, the server will be idle, waiting for another packet from a client

Scroll back up the terminal window and find the line saying "Ready to process requests". Look just below that line, and the following output should be visible:

```
rad_recv: Access-Request packet from host 127.0.0.1 port 62977, id=218,
length=55
User-Name = "bob"
User-Password = "hello"
NAS-IP-Address = 127.0.0.1
```

```
NAS-Port = 0
```

Apart from the line starting with "`NAS-IP-Address`", the last four lines are the same attributes sent by radtest. The IP address in the second-to-last line is a "pretend" parameter added by radtest to indicate an artificial IP address that identifies the location of the switch during the testing process. Since you logged in from localhost, the IP address could be 127.0.0.1, or it could be one of the addresses used by your local system.

Getting PAP working is an important step towards configuring any kind of authentication. If the above example was successful, congratulations on starting to become a RADIUS expert. If the example did not work, then double-check that instructions were followed carefully. See **Chapter 11 - Debugging** on page 80 for a guide on how to debug the server configuration.

The additional lines printed by the server in between the `Access-Request` and `Access-Accept` are also of interest. The exact format will change in minor ways from version to version of FreeRADIUS, but the general style should be similar. A sample of this output is reproduced below:

```
+- entering group authorize {...}
++[preprocess] returns ok
++[chap] returns noop
++[mschap] returns noop
[suffix] No '@' in User-Name = "bob", looking up realm NULL
[suffix] No such realm "NULL"
++[suffix] returns noop
[eap] No EAP-Message, not doing EAP
++[eap] returns noop
++[unix] returns notfound
[files] users: Matched entry bob at line 1
++[files] returns ok
++[expiration] returns noop
++[logintime] returns noop
++[pap] returns updated
Found Auth-Type = PAP
+- entering group PAP {...}
[pap] login attempt with password "hello"
[pap] Using clear text password "hello"
[pap] User authenticated successfully
++[pap] returns ok
+- entering group post-auth {...}
++[exec] returns noop
```

Most of this text can simply be ignored, as most of the entries are not related to this test, but instead relate to the default installation. However, there following few lines in the output do relate to the test:

PAP Authentication Output Meaning	
Output	**Meaning**
`[files] users: Matched entry bob at line 1`	This line shows that the `users` file was read and that the entry added for user "bob" was found as the first line of the file. This entry sets the "known good" password for the user.

`[pap] login attempt with password "hello"`
`[pap] Using clear text password "hello"`
`[pap] User authenticated successfully`
	These lines show that the user tried to log in with password "hello", and that the server was configured to expect the password "hello". Since the passwords match, the user has been authenticated.

Table 5.1.1 PAP Authentication Output Meaning

If a user tries to log in with an incorrect password, then the output will be different. See section **5.0.2 Incorrect Password** on page 26 for information.

5.0.2 Incorrect Password

The Test

The following example tests the server's response when a user inputs the incorrect password. All the configuration from the previous example will be left in place; however, in order to gain familiarity with the server output in the event of an incorrect password, the server will be restarted.

1. In the terminal window where the server is running, press CTRL-C to stop the server.
2. Enter `radiusd -X`.
   Wait for the following text to appear in the terminal window: "Ready to process requests".
3. To send an authentication request packet with the incorrect password ("goodbye") to the server, type the following command in another terminal window on the same machine:

```
$ radtest -x bob goodbye 127.0.0.1 0 testing123
```

If the authentication process worked correctly, the server will return an `Access-Reject` message after a second or two, and the window with radtest will display the following (or something similar):

```
rad_recv: Access-Reject packet from host 127.0.0.1 port 1812, id=110,
length=20
```

The above message indicates that authentication failed, as expected.

The Server Output

The reasons that the user's request has been rejected are contained in the server output. To view these reasons, start by switching back to the terminal window where the server is running and look at the final lines. They should be similar to the following:

```
Delaying reject of request 0 for 1 seconds
Going to the next request
```

```
Waking up in 0.9 seconds.
Sending delayed reject for request 0
Sending Access-Reject of id 110 to 127.0.0.1 port 65457
Waking up in 4.9 seconds.
```

Collectively, these lines show that the server rejected the user.

The first line, above, indicates that the reject message was delayed for one second. This delay is for security reasons. It is generally a good practice to delay rejects for a short period so as to prevent "brute force" attempts, wherein attackers perform tens of thousands of password checks per second.

Now scroll back up the terminal window, and find the line that states "Ready to process requests". Look just below it, and something similar to the following should appear:

```
rad_recv: Access-Request packet from host 127.0.0.1 port 65457, id=110,
length=55
User-Name = "bob"
User-Password = "goodbye"
NAS-IP-Address = 127.0.0.1
NAS-Port = 0
```

As expected, the output above reflects the parameters passed via radtest.

The interesting part of the output is contained in the portion in between the `Access-Request` and `Access-Reject`. It is reproduced below.

```
+- entering group authorize {...}
++[preprocess] returns ok
++[chap] returns noop
++[mschap] returns noop
[suffix] No '@' in User-Name = "bob", looking up realm NULL
[suffix] No such realm "NULL"
++[suffix] returns noop
[eap] No EAP-Message, not doing EAP
++[eap] returns noop
++[unix] returns notfound
[files] users: Matched entry bob at line 1
++[files] returns ok
++[expiration] returns noop
++[logintime] returns noop
++[pap] returns updated
Found Auth-Type = PAP
+- entering group PAP {...}
[pap] login attempt with password "goodbye"
[pap] Using clear text password "hello"
[pap] Passwords don't match
++[pap] returns reject
Failed to authenticate the user.
Using Post-Auth-Type Reject
+- entering group REJECT {...}
[attr_filter.access_reject] expand: %{User-Name} -> bob
attr_filter: Matched entry DEFAULT at line 11
++[attr_filter.access_reject] returns updated
```

The following line, found in the above output, shows that the user "bob" is indeed listed in the `users` file:

```
[files] users: Matched entry bob at line 1
```

This output shows that, when the `users` file was read by the server, an entry for user "bob" was found in the first line. The entry in the `users` file also lists the "known good" password for that user.

Even though the password in the `users` file ("hello") differs from the password entered in the terminal window ("goodbye"), it is important to note that the entry is still matched by the user's name ("bob"). This matching occurs because, at this point in the process, the password entered in the terminal window and the "known good" password have not been compared by the server.

The password comparison can be seen later in the output:

```
[pap] login attempt with password "goodbye"
[pap] Using clear text password "hello"
[pap] Passwords don't match
```

These lines show that the user tried to log in with a password ("goodbye") and that the server was configured to expect a different password ("hello"). Since the passwords do not match, the user has been rejected.

In these examples, the server output is fairly short and easy to read. When the output is longer and more complicated, finding the important pieces can be difficult. The best approach is to look for the words WARNING, ERROR, REJECT, or FAILED in the output. There will often be helpful text describing what went wrong and how to fix the problem.

With practice, reading the server output becomes easier. Much of the effort involved is little more than scanning the output for key words.

5.0.3 Incorrect Shared Secret

The "shared secret" in RADIUS is like a key that is shared between the client and server. It is used to sign packets to prevent forgery and to hide secret data such as passwords.

The Test

In the example below, authentication is attempted using an incorrect "shared secret". All the configuration from the previous example will remain the same in this example. The server is first restarted, and then the test is run, as follows:

1. In the terminal window where the server is running, press CTRL-C to stop the server and type `radiusd -X`. Wait for the text saying "Ready to process requests".
2. In another terminal window on the same machine, type the following command to send an authentication request packet to the server, as before but with the shared secret set to "testingXYZ":

```
$ radtest -x bob hello 127.0.0.1 0 testingXYZ
```

If all goes well, the server will return an `Access-Reject` message after a second or two, and the window with radtest will print something similar to the following:

```
rad_recv: Access-Reject packet from host 127.0.0.1 port 1812, id=25,
length=20
rad_verify: Received Access-Reject packet from client 127.0.0.1 port
1812 with
invalid signature! (Shared secret is incorrect.)
```

This text means that authentication failed, as expected. The error message also contains a suggestion as to why the authentication failed in the final line: `Shared secret is incorrect.`

The Server Output

The server output contains the reasons that the user's request was rejected. To view these reasons, switch back to the terminal window where the server is running and look at the final lines. They should look similar to the following:

```
Delaying reject of request 0 for 1 seconds
Going to the next request
Waking up in 0.9 seconds.
Sending delayed reject for request 0
Sending Access-Reject of id 25 to 127.0.0.1 port 57036
Waking up in 4.9 seconds.
```

These lines show that the server rejected the user's request, as before. Next, scroll back up the terminal window and find the line saying `Ready to process requests`. Just below this line, the following will appear:

```
rad_recv: Access-Request packet from host 127.0.0.1 port 57036, id=25,
length=55
User-Name = "bob"
User-Password = "\310vR\007\016\253\237\201\\\t}\332u\017("
NAS-IP-Address = 127.0.0.1
NAS-Port = 0
```

This output does not reflect the parameters passed via radtest. Because the "shared secret" is used to decode the password, using an incorrect "shared secret" means that the password will be decoded to the wrong value. In the fourth line above, the password appears to be a string of garbage, instead of the string "hello". Thus, the decoded password will not have the same value as that sent by radtest, because the *incorrect* "shared secret" was used to decode the *correct* password.

As before, the next step is to have a look at the rest of the output between the `Access-Request` and `Access-Reject`. This output is reproduced below:

```
+- entering group authorize {...}
++[preprocess] returns ok
++[chap] returns noop
++[mschap] returns noop
[suffix] No '@' in User-Name = "bob", looking up realm NULL
[suffix] No such realm "NULL"
++[suffix] returns noop
[eap] No EAP-Message, not doing EAP
++[eap] returns noop
++[unix] returns notfound
[files] users: Matched entry bob at line 1
++[files] returns ok
++[expiration] returns noop
++[logintime] returns noop
++[pap] returns updated
Found Auth-Type = PAP
+- entering group PAP {...}
[pap] login attempt with password
"\310vR\007\016\253\237\201\\\t}\332u\017("
[pap] Using clear text password "bob"
[pap] Passwords don't match
++[pap] returns reject
Failed to authenticate the user.
WARNING: Unprintable characters in the password. Double-check the shared
secret
```

```
on the server and the NAS!
Using Post-Auth-Type Reject
+- entering group REJECT {...}
[attr_filter.access_reject] expand: %{User-Name} -> bob
attr_filter: Matched entry DEFAULT at line 11
++[attr_filter.access_reject] returns updated
```

As before, we see that the user "bob" is found in the `users` file, which sets the known good password:

```
[files] users: Matched entry bob at line 1
```

The last few lines of the above output show the reason for the failed authentication, as well as a new WARNING message.

```
[pap] login attempt with password
"\310vR\007\016\253\237\201\\\t}\332u\017("
[pap] Using clear text password "bob"
[pap] Passwords don't match
++[pap] returns reject
Failed to authenticate the user.
WARNING: Unprintable characters in the password. Double-check the shared
secret on the server and the NAS!
```

These lines show that the user tried to log in with a garbage password, and that the server was configured to expect the password "hello". Since the passwords do not match upon being decoded using the incorrect "shared secret", the user has been rejected.

The most useful line here is the WARNING message. It is a guess from the server as to what has gone wrong. It instructs you to check the secrets on the client and server configuration. The best way to fix the secrets is to re-enter both of them. **Use a text editor to copy the secret from a text file and to paste the same text into both the client and the server configuration.** This practice is better than trying to check the secrets by eye, since hidden characters may be used or two different characters that look very similar may be interchanged.

When that WARNING message appears, it is always recommended recommend that the shared secrets be re-entered. The most common cause of this WARNING message appearing is that the client and the server contain two different secrets.

The only other cause of this WARNING message is a programming error on the client and/or server, which means that software has a bug, and does not correctly implement the RADIUS protocol. Of the two situations, the highest probability is that an administrator mistyped a shared secret.

5.0.4 Unknown Users

In the example below, authentication is attempted for a user who is unknown to the server.

The Test

All of the configuration from the previous example will remain the same. First, the server will be restarted, as follows:

1. In the terminal window where the server is running, press CTRL-C to stop the server.
2. Type `radiusd -X`.
   Wait for the text saying Ready to process requests.
3. In another terminal window on the same machine, type the following command:

```
$ radtest doug hello 127.0.0.1 0 testing123
```

This command sends an authentication request packet to the server, with the following parameters:

- Name doug
- Password hello
- To server 127.0.0.1
- Pretending to log in on switch port 0
- Using a shared secret of testing123

The Server Output

If all goes well, you should see the server returning an `Access-Reject` message after a second or two, and the window with radtest should print something similar to the following:

```
rad_recv: Access-Reject packet from host 127.0.0.1 port 1812, id=213,
length=20
```

The above text shows that authentication failed, as expected. As before, the reasons for the user being rejected are contained in the server output. Switch back to the terminal window where the server is running, and look at the final lines. They should contain a now familiar message, as follows:

```
Sending delayed reject.
```

Scroll back up the terminal window and look for the following `Access-Request` packet:

```
rad_recv: Access-Request packet from host 127.0.0.1 port 59032, id=213,
length=56
User-Name = "doug"
User-Password = "hello"
NAS-IP-Address = 10.10.0.110
NAS-Port = 0
```

This output reflects the parameters that were entered via radtest, including the password. Look again at the rest of the output between the `Access-Request` and `Access-Reject`, as reproduced below:

```
+- entering group authorize {...}
++[preprocess] returns ok
++[chap] returns noop
++[mschap] returns noop
[suffix] No '@' in User-Name = "doug", looking up realm NULL
[suffix] No such realm "NULL"
++[suffix] returns noop
[eap] No EAP-Message, not doing EAP
++[eap] returns noop
++[unix] returns notfound
++[files] returns noop
++[expiration] returns noop
++[logintime] returns noop
[pap] WARNING! No "known good" password found for the user.
Authentication may fail
because of this.
++[pap] returns noop
No authenticate method (Auth-Type) found for the request: Rejecting the
```

```
user
Failed to authenticate the user.
Login incorrect: [doug] (from client localhost port 0)
Using Post-Auth-Type Reject
+- entering group REJECT {...}
[attr_filter.access_reject] expand: %{User-Name} -> doug
attr_filter: Matched entry DEFAULT at line 11
++[attr_filter.access_reject] returns updated
```

Unlike in the example with user "bob", an entry for user "doug" was not found in the `users` file. If the user "doug" were found in the `users` file, the text above would have included a line saying [files] users: Matched entry doug, which is does not. The only reference to [files] is:

```
[files] returns noop
```

This line shows that there was no matching entry found in the `users` file and that no operation was performed. To understand why the authentication may have failed, continue reading the output until a new WARNING message is found:

```
[pap] WARNING! No "known good" password found for the user.
Authentication may fail because of this.
```

These lines show that the server does not know how to authenticate the user "doug". Since there is no known good password that could be used as the basis for a comparison, the user has been rejected.

There is another useful message in the output:

```
No authenticate method (Auth-Type) found for the request: Rejecting the
user
```

This message is less clear than the WARNING message. It means that the server has been unable to determine which authentication protocol it should use for authenticating the user. In contrast, the output from the previous tests had the following text:

```
++[pap] returns updated
Found Auth-Type = PAP
+- entering group PAP {...}
[pap] login attempt with password "hello"
```

The text above shows that the `pap` module saw both the known good password, which was set by the `users` file, and the User-Password attribute in the request. The module then determined that the PAP protocol should be used for authentication. When there is no known good password, the `pap` module will not set `Auth-Type = PAP`, even if the request contains a User-Password.

Again, the useful line here is the WARNING message. It is a guess from the server as to what has gone wrong. It suggests that the user and/or a known good password for that user should be configured. Authentication will continue to fail until both the user and the password are known to the server.

5.1 Password Formats

In previous sections, terminology such as "known good", "Cleartext", and "User" were employed as descriptors to distinguish between different types of passwords. These differences will now be explored in more detail, starting with a study of plain-text passwords and moving on to the wide variety of available password formats.

5.1.1 Plain-text

A plain-text password is stored as a string that is comprehensible for humans. There are two different types of plain-text passwords, as described below.

Older versions of FreeRADIUS (version 1 and earlier) did not distinguish between the two types of passwords. In hindsight, this mistake has been replicated in many third-party How-To's and other documentation.

1. **`User-Password`**
   A password entered by a user into a GUI or password prompt. This password is then placed into an `Access-Request` packet by a client and is sent to a server.

2. **`Cleartext-Password`**
   A password entered by an administrator into a user management system such as the `users` file or a database such as LDAP or SQL. This password is also called the "known good" password, as it is the one known by the administrator to be the correct or good password for the user.

When the server receives a `User-Password` in an `Access-Request`, it looks up the user's name in the management system in order to find the `Cleartext-Password`. The two passwords are then compared. If they are the same, the user is authenticated. If they are different, the user is rejected.

It follows that `User-Password` should never be placed into a user management system, nor should it appear in the `users` file or in any database such as LDAP or SQL. Although placing `User-Password` into a user management system or other inappropriate location as described above may work in some select situations, in the vast majority of situations it will cause problems. Using `User-Password` in this manner will render the server less capable than using the more appropriate `Cleartext-Password` in these locations.

5.1.2 Hashed Passwords

There exists a perception that storing plain-text passwords poses a security hazard. To address this security concern, people started using hashed passwords. Hashed passwords are created by taking a clear text string and performing an algorithm on it to get a completely different value. This value is the same every time; thus, the hashed password can be stored in a database and checked against the user's entered password.

5.1.2.1 Crypt

One well-known form of hashed passwords is the Unix `crypt` format. The `crypt` password can be used for authentication instead of the plain-text password.

The `crypt` passwords are generated by use of the `radcrypt` program, which is only available in versions 2.1.10 and later. For systems that use earlier versions of FreeRADIUS, the tests in this section can be skipped.

The `crypt` format involves taking a random string, called a "salt", and using the password as a key to the DES encryption algorithm. The algorithm encrypts a constant string with a combination of the password and the "salt"; the output of this process is called a `crypt` password.

This method has a number of security benefits. For example, while it is difficult to turn the initial output of `crypt` back into the plain-text password, the complexity added by use of the "salt" means that every plain-text password can map to 4096 or more `crypt` passwords.

The following example shows the random salt process in action, as the `radcrypt` program is run multiple times:

```
$ radcrypt --des hello
```

```
0J.YPfV/j4qPA

$ radcrypt --des hello
5uDzcwJ44fgkM

$ radcrypt --des hello
IUQCaQFeUSvyE
```

The above messages show that the same input password results in many different output passwords. The output passwords will start to repeat themselves, however, if the program is run many more times.

The Test

Below, the tests performed in 5.0.1 Correct User and Password on page XXX are repeated, except this time the entry in the `users` file is changed to the following:

```
bob Crypt-Password := "0J.YPfV/j4qPA"
```

This text must be the first line in the `users` file, and the other line containing `Cleartext-Password` must be deleted. Start the server using `radiusd -X` as before, and run an authentication test:

```
$ radtest bob hello 127.0.0.1 0 testing123
```

The Server Output

The `Access-Accept` being returned by the server should now be visible. The main change in the server output from the previous tests using `Cleartext-Password` is the following:

```
[pap] login attempt with password "hello"
[pap] Using CRYPT encryption.
[pap] User authenticated successfully
```

The output above shows that the server is now using crypt authentication. The PAP module authenticates the user by taking the contents of the `User-Password` and creating a crypt version. It then compares that version to the contents of the `Crypt-Password`. In the above example they match, so the user is authenticated.

Another version of the `crypt` password shown above can be accomplished by following these instructions: Edit the `users` file and change the contents of the `Crypt-Password` to the text printed by the second run of `radcrypt`:

```
bob Crypt-Password := "5uDzcwJ44fgkM"
```

When `radtest` is run as before, another `Access-Accept` should be returned.

As an exercise, try changing the password given to `radtest` and see how the server behaves. Look for the key lines in the server output, as highlighted earlier.

To further ensure that encryption is working correctly, try editing the `users` file to create an incorrect `crypt` password, such as by changing the `5` above to a `6`.

5.1.2.2 NT

Another form of a hashed password is what is often called the `NT-hash` or `NT-Password`. This format is most commonly used in Microsoft Windows environments, with Active Directory or Samba. Unlike the

Unix `crypt` authentication, it does not use a random salt string. The following example shows when the `smbencrypt` program is run (without the use of a random salt string):

```
$ smbencrypt hello
LM Hash                                  NT Hash
---------------------------------    ---------------------------------
FDA95FBECA288D44AAD3B435B51404EE    066DDFD4EF0E9CD7C256FE77191EF43C

$ smbencrypt hello
LM Hash                                  NT Hash
---------------------------------    ---------------------------------
FDA95FBECA288D44AAD3B435B51404EE    066DDFD4EF0E9CD7C256FE77191EF43C
```

Unlike the `radcrypt` tests, the output of `smbencrypt` is the same across two different runs of the `smbencrypt` program. This repetitive behavior means that it is much easier to turn an NT hash back into a plain-text password.

The Test

To authenticate the user by employing NT-hash passwords, edit the `users` file and change the entry for user "bob" to the following (ensure that there are no other entries left for bob in the `users` file, as those could change the results of the test):

```
bob NT-Password := "066DDFD4EF0E9CD7C256FE77191EF43C"
```

When `radtest` is then run as before, another `Access-Accept` should be returned. To see how the client and server behave in a variety of situations, try changing the hex string for `NT-Password` and the password used by `radtest`.

The Server Output

Take the time to look carefully at the server output to see the differences from the previous tests. A sample is given below:

```
[pap] login attempt with password "hello"
[pap] Using NT encryption.
[pap] expand: %{User-Password} -> hello
[pap] NT-Hash of hello = 066ddfd4ef0e9cd7c256fe77191ef43c
[pap] expand: %{mschap:NT-Hash %{User-Password}} ->
066ddfd4ef0e9cd7c256fe77191ef43c
[pap] User authenticated successfully
```

This output is quite a bit more complicated than the output from the previous tests; however, it shows exactly how the server is authenticating the user. When the PAP module receives the password `hello`, it calculates the `NT-Hash` of the password. The result of the calculation is the hex string `066ddfd4ef0e9cd7c256fe77191ef43c`, which is the lower-case version of the output of `smbencrypt`. The server compares the two strings, sees that they are identical, and authenticates the user.

5.1.2.3 Generic Hashed

In addition to the specific types of hashed passwords mentioned in the previous sections, there are many additional kinds of hashed passwords, each with their own properties. FreeRADIUS supports most common, and a few uncommon, types. The complete list is given below, with associated explanations.

- Cleartext-Password
  The "plain text" password.
- Crypt-Password
  The Unix crypt password. There are a number of variations, which are often system specific. FreeRADIUS relies on local libraries to understand these passwords; therefore, copying the passwords from one machine to another may result in authentication failures.
- LM-Password
  Microsoft "Lan Manager" password. Supported only for historical reasons, it should never be used.
- MD5-Password
  The MD5 hash of the password. This hash type does not use a salt and is supported only for compatibility reasons. It should not be used.
- NS-MTA-MD5-Password
  A Netscape-specific hash of the password. Supported only for historical reasons, it should not be used.
- NT-Password
  The MicrosoftWindows "NT" hash of the password. This hash type does not use a salt and is supported for compatibility reasons.
- SHA-Password
  The SHA1 hash of the password. This hash type does not use a salt and is supported only for compatibility reasons. It should not be used.
- SMD5-Password
  A salted MD5 hash of the password. On certain systems, this format may be the same as that used by crypt.
- SSHA-Password
  A salted SHA1 hash of the password. On certain systems, this format may be the same as that used by crypt.

All of these passwords can be used interchangeably in the examples shown above. Just replace `Cleartext-Password` with one of the above names, along with a value appropriate for the named hashing method. For example, the MD5 hash of `hello` is `b1946ac92492d2347c6235b4d2611184`.

We recommend using the `Cleartext-Password` whenever possible. A secondary choice would be the `NT-Password`. All other formats should be avoided, as they are compatible only with PAP authentication and are not compatible with all other authentication methods.

There are security implications inherent in storing plain-text passwords. For example, if an attacker successfully steals a copy of a database containing plain-text passwords, they would then have complete access to all of the user names and passwords. This type of attack was historically possible in Unix systems, where end users would log into the machine holding a copy of the `/etc/passwd` file. They are also possible in a Microsoft Windows environment, where the Active Directory server is directly accessible to end users. These attacks are, however, much less likely in a properly configured RADIUS system.

When using plain-text passwords, therefore, proper steps must be taken to secure the RADIUS system, as outlined in the checklist below:

1. Only administrators should have login access to the machine running RADIUS.
2. Only administrators should have login access to the machine running the database(s)

3. Only administrators should have read access to the database(s)
4. The RADIUS server and databases should be on an administrative network, and inaccessible by end users.
5. The RADIUS server and databases should run minimal services.

In many cases, business realities outweigh security concerns. Plain-text passwords are required for many authentication protocols. Often, there is no other choice but to store plain-text passwords in order to authenticate users. So long as proper security procedures such as those outlined above are followed, storing plain-text passwords in a RADIUS system is a safe option.

5.2 Alternate Authentication Methods

This section details alternate simple authentication methods. These methods are more complicated than PAP, but are simpler than methods that require certificates or public / private key combinations. The sections below require that the `users` file has an entry at the top for the test user `bob`, as outlined above in Section 5.1XXX.

```
bob Cleartext-Password := "hello"
```

Any other entries for user `bob` should be deleted.

The examples given in this section require the use of the client tools `radtest`, `radclient`, and `radeapclient`, from version 2.1.10 or later. The client tools installed on servers using earlier versions will not be appropriate. To follow the examples given in this section, a more recent version must be installed in a local directory (e.g. `~/freeradius`) and the tools from that directory utilized. The newer client tools will still communicate with older versions of the server or with any RADIUS server. The only difference between the versions is that the newer tools are much easier to use.

5.2.1 CHAP

The Challenge-Handshake Authentication Protocol, or CHAP, is an authentication mechanism that does not send a password to the server. Instead, the client creates a random string, called the `challenge`, and performs an MD5 hash to combine the challenge with the password. The client then sends both the challenge and the hash to the server. If the server has access to the `Cleartext-Password`, it then performs the same MD5 calculation and compares its hash to the one sent by the client.

The benefit of the CHAP method is that the password is never sent in a packet. The drawbacks, however, are many: if the server does not have access to the `Cleartext-Password`, authentication will fail; the RADIUS packets containing CHAP can be trivially replayed, which is a security issue; and the client has no idea whether the server checked the hash or never checked the hash and returned Access-Accept. For these reasons, CHAP should not be used for new deployments.

That being said, it is still in wide-spread use and is a vital part of many networks.

The Test

We can test CHAP by use of the `radtest` command:

```
$ radtest -x -t chap bob hello 127.0.0.1 0 testing123
```

This command line is similar to the ones in the previous section, with the addition of `-t chap`, which tells `radtest` to perform CHAP authentication.

The Server Output

The output generated from this command will be similar to the following:

```
Sending Access-Request of id 232 to 127.0.0.1 port 1812
User-Name = "bob"
CHAP-Password := 0xe834861ac62185fa5f1b610ef335df8b3b
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=232,
length=20
```

Notice that the `CHAP-Password` sent looks like a random set of hex digits, rather than the original password `hello`.

5.2.2 MS-CHAP

The Microsoft (MS) CHAP mechanism is similar to the CHAP method and comes in two forms: MS-CHAPv1 and MS-CHAPv2.

MS-CHAPv1 employs the MD4 algorithm instead of MD5, which allows it to use the `NT-Hash` form of the password, in addition to the `Cleartext-Password` version.

MS-CHAPv2 adds an additional challenge-response step between the client and server. This challenge-response step allows the client and server to each validate that the other knows the password. This additional step mean that MS-CHAP is more secure and easier to deploy than normal CHAP.

The Test

Radtest is run again, this time using `-t mschap` in order to request MS-CHAPv1 authentication:

```
$ radtest -x -t mschap bob hello 127.0.0.1 0 testing123
```

The Server Output

The corresponding output text will be similar to:

```
Sending Access-Request of id 79 to 127.0.0.1 port 1812
      User-Name = "bob"
      MS-CHAP-Challenge = 0x8804cc07e901067d
      MS-CHAP-Response =
0x000100000000000000000000000000000000000000000000004976cc59820ac2bfda
b36c9d6de3e6292a025aa34a1a2a02
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=79,
length=84
      MS-CHAP-MPPE-Keys =
0xfda95fbeca288d44ac0782e2de2337dee40e54ee732c1af50000000000000000
      MS-MPPE-Encryption-Policy = 0x00000001
      MS-MPPE-Encryption-Types = 0x00000006
```

Notice that the Access-Request contains an MS-CHAP challenge and response, rather than the original password `hello`. The response also includes some special Microsoft attributes, which can be ignored for now.

This MS-CHAP test should always be done prior to performing PEAP authentication.

5.2.3 EAP-MD5

The EAP-MD5 authentication method is essentially CHAP wrapped in another layer called the Extensible Authentication Protocol (EAP). The EAP layer is most commonly used for wireless authentication. However, the EAP-MD5 type does not provide for encryption keys needed in wireless networks, so it should only be used in wired (i.e. ethernet) environments.

The Test

As before, the `radtest` program is run:

```
$ radtest -x -t eap-md5 bob hello 127.0.0.1 0 testing123
```

The Server Output

The output should be similar to the following:

```
Sending Access-Request packet to host 127.0.0.1 port 1812, id=19,
length=0
      User-Name = "bob"
      Cleartext-Password = "hello"
      NAS-IP-Address = 127.0.0.1
      EAP-Code = Response
      EAP-Type-Identity = "bob"
      Message-Authenticator = 0x00
      NAS-Port = 0
      EAP-Message = 0x0212000801626f62
Received Access-Challenge packet from host 127.0.0.1 port 1812, id=19,
length=80
      ...
Sending Access-Request packet to host 127.0.0.1 port 1812, id=20,
length=65
      ...
Received Access-Accept packet from host 127.0.0.1 port 1812, id=20,
length=49
      EAP-Message = 0x03130004
      Message-Authenticator = 0x0420adbc245c3c5679c77e9b8b9ce08e
      User-Name = "bob"
      EAP-Id = 19
      EAP-Code = Success
```

5.3 Forcing Authentication Method

As seen earlier in this chapter, the client chooses the authentication method to use and the server either accepts or rejects the request. Even if the password is correct, the server can choose to reject the request if the presented authentication method is not allowed by the server.

Instead of rejecting all types of authentication that are not allowed by the server, the administrator would force a particular authentication type, which then results in all other types being rejected.

5.3.1 Issues With Forcing Authentication

The major reason not to force a particular authentication method is that doing so will generally "break" every single other authentication method; in other words, forcing the use of one method means that it is impossible for the end user's system to use any other method.

The most common problem seen when forcing `Auth-Type` is authentication rejection when the end user's system is using one authentication method, while the server administrator has enforced a different method. When this occurs, the following message will appear in the debug log:

```
Attribute ... is required for authentication
```

In this instance, there are only a few possible courses of action:

1. Remove the forced setting of `Auth-Type` so that other authentication methods can be used.
2. Reconfigure the end user system to use the desired authentication method
3. Live with the fact that users will be rejected for not using the desired authentication method.

There is no workaround other than the three items above. Most authentication methods are designed to work correctly or not at all, and, thus, there is no method to make them work in different situations.

5.3.2 Forcing Auth-Type Reject

It is always possible to reject a user, no matter the authentication method they are using. The ability to reject a user for any reason at any time is a requirement of all authentication systems.

For example, the following entry in the `users` file will cause the user `bob` to be rejected, no matter the password entered:

```
bob Auth-Type := Reject
```

A similar configuration in `unlang` is the following:

```
authorize {
        ...
        if (User-Name == "bob") {
                reject
        }
        ...
}
```

Note that `Auth-Type` is not set here. Instead, the word `reject` is placed into the configuration. When the server executes that `reject`, processing of the `authorize` section will automatically be stopped and the user rejected.

5.3.3 Forcing Auth-Type Accept

In some cases, it is useful to send an `Access-Accept` even when the user's password is incorrect. These cases most commonly occur when the database is down or unreachable; keeping users happy then becomes more important to administrators than losing small amounts of revenue to erroneously accepted users.

The server can be forced to send an `Access-Accept` by setting `Auth-Type := Accept`. For example, the following entry in the `users` file will cause the user `bob` to be accepted, no matter what password is used:

```
bob Auth-Type := Accept
```

Note that this causes the user to be *accepted*, not *authenticated*: the server sends an `Access-Accept`, but the user might still not end up with network access. Sometimes sending an `Access-Accept` is not enough for the user to be authenticated. In a few cases, there are additional steps that need to be taken before the NAS lets the user on the network.

The following table shows which authentication methods are compatible with setting `Auth-Type = Accept`.

Authentication Method Compatibility	
Authentication Method	**Forcing Accept**
PAP	Yes
CHAP	Yes
MS-CHAP versions 1 and 2	No
EAP	No

Table 5.3.3. Forced `Access-Accept` and authentication methods.

The above table shows that it is not always possible to authenticate a user by forcing an `Accept`; it does not, however, explain why those choices are possible or impossible. For that information, packet traces from earlier in the chapter must be studied. Sample packet traces are given below. This time, everything but the attributes which give the explanation have been removed.

PAP:

```
rad_recv: Access-Request packet from host 127.0.0.1 port 62977, id=218,
length=55
      ...
      User-Password = "hello"
      ...
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=218,
length=20
      ...
```

CHAP:

```
Sending Access-Request of id 232 to 127.0.0.1 port 1812
      ...
      CHAP-Password := ...
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=232,
length=20
      ...
```

MS-CHAP:

```
Sending Access-Request of id 79 to 127.0.0.1 port 1812
      ...
      MS-CHAP-Challenge = ...
      MS-CHAP-Response = ...
rad_recv: Access-Accept packet from host 127.0.0.1 port 1812, id=79,
length=84
      MS-CHAP-MPPE-Keys = ...
      ...
```

EAP:

```
Sending Access-Request packet to host 127.0.0.1 port 1812, id=19,
length=67
      ...
      EAP-Message = 0x0212000801626f62
...
Received Access-Accept packet from host 127.0.0.1 port 1812, id=20,
length=49
      EAP-Message = 0x03130004
      ...
```

Presented in this way, the explanation should be clear: The authentication methods that are compatible with `Auth-Type = Accept` are ones where nothing special needs to be in the `Access-Accept`. The authentication methods that are not compatible with `Auth-Type = Accept` are ones that need to send specific authentication data in the `Access-Accept`.

When authentication data is sent in the `Access-Accept`, the end user's computer analyses that data to check if it is talking to a "good" server. The exact definition of a good server varies by authentication method. The common element is that the end user's system will abort authentication if the data is wrong or is not available. The security design of these methods is such that it is usually impossible to create a fake response. If creating a fake response were possible, an attacker could potentially exploit that ability to make users access an entire fake network or to steal the user's login credentials.

These limitations mean that forcing `Auth-Type = Accept` is possible only in limited and rare circumstances.

5.3.4 Auth-Type LDAP

A common recommendation on third party web sites and How-To's is to force LDAP authentication via the `users` file, as below:

```
DEFAULT Auth-Type := LDAP
```

This configuration will work only when the `Access-Request` contains a `User-Password` attribute, for PAP authentication. The reason for this limitation is that LDAP databases only implement PAP authentication. LDAP databases are not authentication servers and do not implement CHAP, MS-CHAP, or EAP.

In general, databases should be used as databases and not as authentication servers. The server should request the known good password from the LDAP database. The server can then use this password to perform any one of the supported authentication methods.

5.3.5 Forcing Other Auth-Types

There are few situations where forcing `Auth-Type` will work as expected. In general, forcing `Auth-Type` is only employed when there is a requirement that the user be authenticated via a particular method. As noted earlier, forcing `Auth-Type` also means rejecting the user when they try any other authentication method.

Forcing an authentication method may, however, be useful in some esoteric situations. It should be done only when the configuration in the `Auth-Type` block implements the authentication method used by the end user. In practice, this means that it will work for PAP, sometimes for MS-CHAP, and usually via `ntlm_auth`. It will rarely work for any other authentication method.

For example, a Perl script may use the `User-Password` attribute to perform custom authentication. The script can extend the functionality of the server without requiring source code changes. Similarly, forcing `ntlm_auth` for MS-CHAP requests will work, because the `ntlm_auth` program understands MS-CHAP authentication.

In summary, forcing an authentication method should be done rarely, if ever. It is nearly always best to allow the server to determine the authentication method on its own.