RADIUS Invalid Authenticator and Message-Authenticator Troubleshooting Guide

Contents

Introduction

Authenticator Header

- Authentication of Response
  - When to expect validation failure:
- Password Hiding
- Retransmissions
- Accounting

Message-Authenticator Attribute

- When to use the Message-Authenticator be used:
- When to expect validation failure:
- Validate the Message-Authenticator Attribute

Related Information

Introduction

This document describes RADIUS Authenticator Header and Message-Authenticator attribute, how they are used, and when to expect validation failure.

Authenticator Header

Per RFC 2865, the Authenticator Header is 16 bytes long. When it is used in an Access-Request, it is called a Request Authenticator. When it is used in any kind of response, it is called a Response Authenticator. It is used for:

- Authentication of response
- Password hiding

Authentication of Response

If the server responds with the correct Response Authenticator, the client can compute if that response was related to a valid request.

The client sends the request with the random Authenticator Header. Then, the server that sends the response calculates the Response Authenticator with the use of the request packet along with the shared secret:

```
ResponseAuth = MD5(Code + ID + Length + RequestAuth + Attributes + Secret)
```

The client that receives the response performs the same operation. If the result is the same, the packet is correct.

Note: The attacker that knows the secret value is not able to spoof the response unless it is able to sniff the request.

When to expect validation failure:

Validation failure occurs if the switch does not cache the request anymore (for example, because of timeout). You can also experience it when the shared secret is invalid (yes - Access-Reject also includes this header). This way, the Network Access Device (NAD) can detect the shared secret mismatch. Usually it is reported by Authentication, Authorization, and Accounting (AAA) clients/servers as a shared key mismatch, but it does not reveal the details.

Password Hiding

The Authenticator Header is also used in order to avoid sending the User-Password attribute in plain text. First the Message Digest 5 (MD5 - secret, authenticator) is computed. Then several XOR operations with the chunks of the password are executed. This method is susceptible for offline attacks (rainbow tables) because MD5 is not perceived as a strong one-way algorithm anymore.

Here is the Python script that computes the User-Password:

```
def Encrypt_Pass(password, authenticator, secret):
        m = md5()
        m.update(secret+authenticator)
        return "".join(chr(ord(x) ^ ord(y)) for x, y in zip(password.ljust
       (16,'\0')[:16], m.digest()[:16]))
```

Retransmissions

If any of the attributes in the RADIUS Access-Request have changed (like the RADIUS ID, User-Name, and so on), the new Authenticator field must be generated and all other fields that depend on it must be recomputed. If this is a retransmission, nothing changes.

Accounting

The meaning of the Authenticator Header is different for an Access-Request and an Accounting-Request.

For an Access-Request, the Authenticator is generated randomly and it is expected to receive a response with the ResponseAuthenticator calculated correctly, which proves that the response was related to that specific request.

For an Accounting-Request, the Authenticator is not random, but it is calculated (as per RFC 2866):

```
RequestAuth = MD5(Code + ID + Length + 16 zero octets + Attributes + Secret)
```

This way, the server can check the accounting message immediately and drop the packet if the recalculated value does not match the Authenticator value. The Identity Services Engine (ISE) returns:

```
11038 RADIUS Accounting-Request header contains invalid Authenticator field
```

The typical reason for this is the incorrect shared secret key.

Message-Authenticator Attribute

The Message-Authenticator attribute is the RADIUS attribute defined in RFC 3579. It is used for a similar purpose: to sign and validate. But this time, it is not used in order to validate a response but a request.

The client that sends an Access-Request (it can also be a server that responds with an Access-Challenge) computes the Hash-Based Message Authentication Code (HMAC)-MD5 from its own packet, and then adds the Message-Authenticator attribute as a signature. Then, the server is able to verify it performs the same operation.

The formula looks similar to the Authenticator Header:

```
Message-Authenticator = HMAC-MD5 (Type, Identifier, Length, Request Authenticator,
Attributes)
```

The HMAC-MD5 function takes in two arguments:

- The payload of the packet, which includes the 16 byte Message-Authenticator field filled with zeros
- The shared secret

When to use the Message-Authenticator be used:

The Message-Authenticator must be used for every packet, which includes the Extensible Authentication Protocol (EAP) message (RFC 3579). This includes both the client that sends the Access-Request and the server that responds with the Access-Challenge. The other side silently drops the packet if validation fails.

When to expect validation failure:

Validation failure occurs when the shared secret is invalid. Then, the AAA server is not able to validate the request.

The ISE reports:

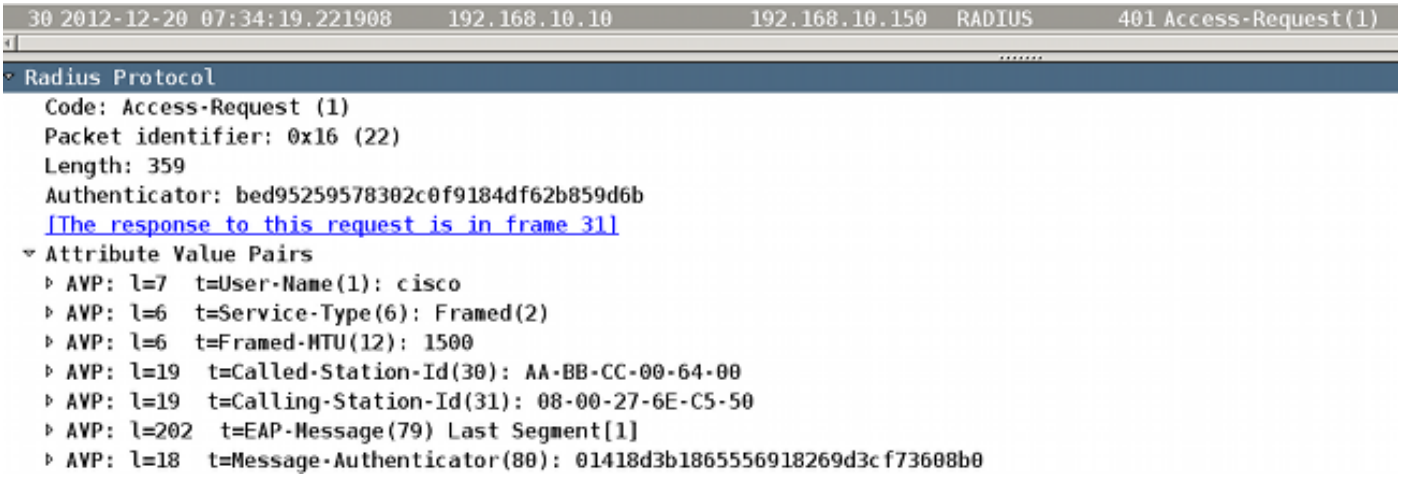
```
11036 The Message-Authenticator Radius Attribute is invalid.
```

This usually occurs at the later stage when the EAP message is attached. The first RADIUS packet of the 802.1x session does not include the EAP message; there is no Message-Authenticator field and it is not possible to verify the request, but at that stage, the client is able to validate the response with the use of the Authenticator field.

Validate the Message-Authenticator Attribute

Here is an example to illustrate how you manually count the value in order to make sure it is computed correctly.

Packet number 30 (Access-Request) has been chosen. It is in the middle of the EAP session, and the packet includes the Message-Authenticator field. The aim is to verify that the Message-Authenticator is correct:



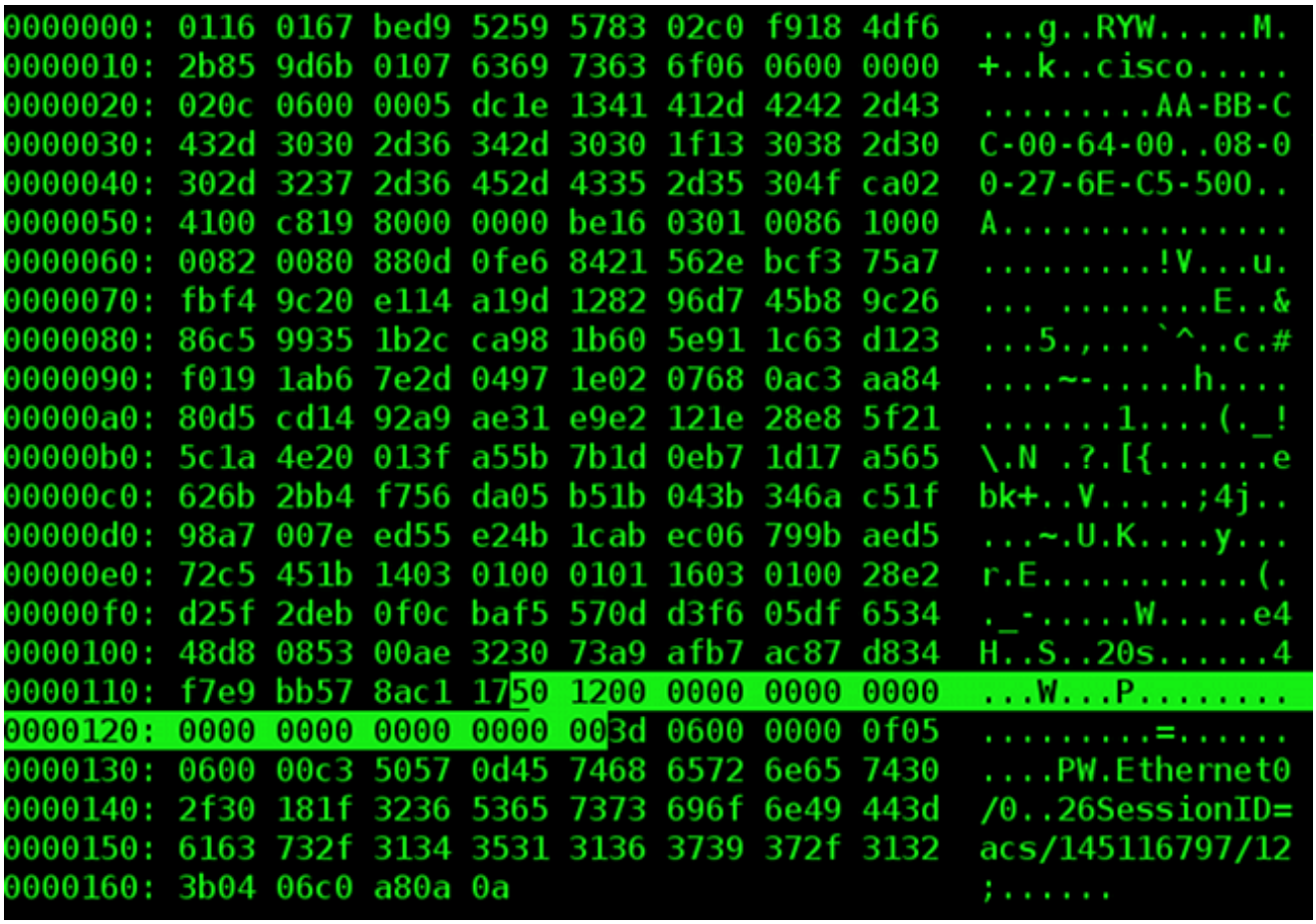
1. Right-click **Radius Protocol** and choose **Export selected packet bytes**.

2. Write that RADIUS payload to a file (binary data).

3. In order to compute Message-Authenticator field, you must put zeros there and compute the HMAC-MD5.

   For example, when you use hex/binary editor, such as vim, after you type ":%!xxd", which switches to hex mode and zeroes 16 bytes starting after "5012" (50hex is 80 in dec which is Message-Authenticator type, and 12 is the size which is 18 including the Attribute Value Pairs (AVP) header):

   Please note that vim and other text editors can add an extra LineFeed character ('0a' in hex) to the end of the file upon saving. Make sure to use the binary mode of your text editor (vim -b file) to prevent the LineFeed character from appearing.



After that modification, the payload is ready. It is necessary to return back to hex/binary mode (type: ":%!xxd -r") and save the file (":wq"). You can double-check an extra LineFeed character was not added by utilizing 'hexdump -C file'.

4. Use OpenSSL in order to compute HMAC-MD5:

<#root>

```
pluton # cat packet30-clear-msgauth.bin | openssl dgst -md5 -hmac 'cisco'
(stdin)=
```

01418d3b1865556918269d3cf73608b0

The HMAD-MD5 function takes two arguments: the first one from standard input (stdin) is the message itself and the second one is the shared secret (Cisco in this example). The result is exactly the same value as the Message-Authenticator attached to the RADIUS Access-Request packet.

The same can be computed with the use of the Python script:

Python 2:

<#root>

```
pluton # cat hmac.py
```

```
#!/usr/bin/env python

import base64
import hmac
import hashlib

f = open('packet30-clear-msgauth.bin', 'rb')
try:
    body = f.read()
finally:
    f.close()

digest = hmac.new('cisco', body, hashlib.md5)
d=digest.hexdigest()
print d

pluton # python hmac.py
```

01418d3b1865556918269d3cf73608b0

Python 3:

```
import hmac
import hashlib

with open('packet30-clear-msgauth.bin', 'rb') as f:
    body = f.read()

digest = hmac.new(b'cisco', body, hashlib.md5)
print(digest.hexdigest())
```

The previous example presents how to calculate the Message-Authenticator field from the Access-Request. For Access-Challenge, Access-Accept, and Access-Reject, the logic is exactly the same, but it is important to remember that Request Authenticator must be used, which is provided in the previous Access-Request packet.

Related Information

- **RFC 2865**
- **RFC 2866**
- **RFC 3579**
- **Technical Support & Documentation - Cisco Systems**