Modular MSSP NAC Architecture Using FreeRADIUS V2

Author: Ege Bulut
Version: 2.0
Date: 07.04.2025

Purpose

This module implements **policy-driven RADIUS authorization and authentication** using FreeRADIUS `rlm_python3`, enabling:

- Dynamic policy matching on incoming RADIUS requests
- Flexible selection of authentication sources (LDAP, SQL, etc.)
- Dynamic reply attributes (VLAN assignment, messages, CoA triggers)
- Zero-downtime configuration reloads
- Modular extension for future authentication and action handlers

This approach **decouples policy logic from FreeRADIUS's declarative configuration**, providing maintainability and vendor independence.

Protocol Taxonomy for NAC Authentication

Protocol	Purpose	Password Visibility	Flexibility to Control Auth Logic	Usual Use Case
PAP	Cleartext password exchange	You get cleartext password	Full control, can auth anywhere	Web login, MAC auth fallback
CHAP	Challenge/response (hashed)	No cleartext	Limited (need pre-hashed passwords)	Rare today, older devices
MS-CHAPv2	NTLM challenge/response	No cleartext	Limited (need NT hash or winbind)	VPNs, older Windows clients
EAP-TLS	TLS mutual certificate authentication	You get client certificate	Full control, can auth any cert CA	Corporate devices with certificates

PEAP-MSCHAPv2	TLS tunnel + NTLM challenge/response	No cleartext	Must use winbind for AD auth	Windows domain login, 802.1X wired/wireless
EAP-TTLS/PAP	TLS tunnel + PAP inside	Cleartext inside the tunnel	Full control, same as PAP	Flexible, less common on Windows

Protocol	Needs winbind?	Multiple ADs?	Password visibility	Best for
PAP	No	Yes	Cleartext	Captive portal, fallback
MS-CHAPv2	Yes or NT hash	No (1 domain)	No	VPN, old clients
PEAP-MSCHAPv2	Yes	No (1 domain per inst)	No	Windows 802.1X login
EAP-TLS	No	Yes (multiple CAs)	Cert only	Secure corporate authentication
EAP-TTLS/PAP	No	Yes	Cleartext	Flexible alternative to PEAP

Design Decisions

1. Policy Source

- YAML-based (`policies.yaml`) configuration to define policy rules
- **Pros:**
  - Human-readable
  - Easily version-controlled
  - Decouples configuration from code
  - Different policy sets can be attached S3M engine

2. Auth Sources

- YAML-based (`auth_sources.yaml`) configuration to define connection parameters (e.g., LDAP)
- **Rationale:**
  - Different policies can reuse shared credentials
  - Multiple authentication backends supported

- Centralized secrets management
- Decouples source of truths from authentication logic

3. Policy Matching

- Flexible match conditions:
  - NAS IP address
  - MAC address prefix
  - Username suffix
  - (Extensible and abstracted via match section of `policy_matcher.py`)

4. Authentication

- Policies can:
  - Require no authentication (pure policy routing)
  - Require one or multiple authentication sources
  - Use "AND"/"OR" logic to combine sources (extension via `authenticate.py` for different authentication procedures)

5. Action Handling

- Policies specify:
  - Accept or reject
  - VLAN assignment
  - Reply messages
  - Class attributes
  - Dynamic authorization triggers (`dynamic_authorization.py`)

6. Zero Downtime Reload

- A reload flag file triggers in-process YAML reloads
- **Why:**
  - Avoids `radiusd` restarts
  - Allows frequent policy updates without service interruption

---

5. Module Responsibilities

dispatcher.py

- Entry point for FreeRADIUS calls
- Loads policies and auth sources
- Evaluates match conditions
- Invokes authentication modules
- Sets reply and control attributes
- Routes FreeRADIUS respectively for protocol
- Logs accounting requests

Proof of Concept Code Here:

```
import yaml
import radiusd
import os
from auth_ldap import authenticate_ldap
#from auth_ntlm import authenticate_ntlm
#from auth_server_challenge_parser import parse_server_challenge
#from auth_mschapv2_parser import parse_mschapv2

CONFIG_PATH = "/etc/freeradius/3.0/mods-config/python3/policies.yaml"
POLICIES = []

AUTH_SOURCES_PATH = "/etc/freeradius/3.0/mods-config/python3/auth_sources.yaml"
AUTH_SOURCES = {}

#CHALLENGE_CACHE = {}

RELOAD_FLAG = "/etc/freeradius/3.0/mods-config/python3/.reload_policies"

def load_policies():
    global POLICIES
    radiusd.radlog(radiusd.L_INFO, "Loading policy YAML...")
    with open(CONFIG_PATH, "r") as f:
        data = yaml.safe_load(f)
        POLICIES = data.get("policies", [])
    radiusd.radlog(radiusd.L_INFO, f"Loaded {len(POLICIES)} policies.")

def load_auth_sources():
    global AUTH_SOURCES
    if os.path.exists(AUTH_SOURCES_PATH):
        radiusd.radlog(radiusd.L_INFO, "Loading authentication sources YAML...")
        with open(AUTH_SOURCES_PATH, "r") as f:
            data = yaml.safe_load(f)
            AUTH_SOURCES = data.get("auth_sources", {})
        radiusd.radlog(radiusd.L_INFO, f"Loaded {len(AUTH_SOURCES)} auth sources.")
    else:
        AUTH_SOURCES = {}
        radiusd.radlog(radiusd.L_WARN, "Auth sources YAML not found.")

def instantiate(*args, **kwargs):
    load_policies()
    load_auth_sources()
    return 0

def maybe_reload_configs():
    if os.path.exists(RELOAD_FLAG):

            reply_attrs.append(("Tunnel-Type", "VLAN"))
            reply_attrs.append(("Tunnel-Medium-Type", "IEEE-802"))
            reply_attrs.append(("Tunnel-Private-Group-Id", action["vlan"]))

        if action.get("reject"):
            radiusd.radlog(radiusd.L_INFO, f"{name} policy rejects.")
            return (radiusd.RLM_MODULE_REJECT, (), ())

        # If there's an auth source, attempt authentication
        auth_source_name = matched_policy.get("auth_source")
```

```
        if auth_source_name:
            auth_config = AUTH_SOURCES.get(auth_source_name)
            if not auth_config:
                radiusd.radlog(radiusd.L_ERR, f"Auth source {auth_source_name} not defined.")
                return (radiusd.RLM_MODULE_REJECT, tuple(reply_attrs), ())

            if not cleartext_pw:
                expected_pw = ""
                if auth_config["type"] == "yaml":
                    # Load the local credentials file
                    credentials_path = auth_config["path"]
                    with open(credentials_path, "r") as f:
                        credentials_data = yaml.safe_load(f)
                    credentials = credentials_data.get("credentials", {})

                    expected_pw = credentials.get(user)
                    if not expected_pw:
                        radiusd.radlog(radiusd.L_ERR, f"User '{user}' not found in local credentials.")
                        return (radiusd.RLM_MODULE_REJECT, tuple(reply_attrs), ())

                    # Provide the Cleartext-Password for MSCHAP/MSCHAPv2
                    control_attrs.append(("Cleartext-Password", expected_pw))
                    radiusd.radlog(radiusd.L_INFO, f"Set Cleartext-Password for user '{user}' from local_user_list as '{expected_pw}'.")

                if auth_config["type"] == "sql":
                    # Invoke SQL check python helper code
                    radiusd.radlog(radiusd.L_INFO, "Authentication type: " + auth_config["type"])

                if not eap_message:
                    radiusd.radlog(radiusd.L_INFO, "MSCHAP Found, skipping pyt {parsed}")
#                            # You can now call ntlm_auth with the challenge/response
#                            # (e.g., via subprocess)
#                            # And then generate MPPE keys if successful
#                            nt_response = parsed["nt_response"]
#
#                        except Exception as e:
#                            radiusd.radlog(radiusd.L_ERR, f"Failed to parse MSCHAPv2: {e}")
#
#                        if  server_challenge != "" and nt_response != "":
#                            radiusd.radlog(radiusd.L_INFO, "MSCHAP Challenge and Response Found...")
#                            auth_source_name = matched_policy.get("auth_source")
#                            auth_config = AUTH_SOURCES.get(auth_source_name)
#                            if not auth_config:
#                                radiusd.radlog(radiusd.L_ERR, f"Auth source {auth_source_name} not defined.")
#                                return (radiusd.RLM_MODULE_REJECT, tuple(reply_attrs), ())
#                            username_no_suffix = user.split("@")[0]
#
#                            if authenticate_ntlm(auth_config, username_no_suffix, server_challenge, nt_response):
```

```
#                                radiusd.radlog(radiusd.L_INFO, f"NTLM auth success for {username_no_suffix}")
#                                return (radiusd.RLM_MODULE_OK, tuple(reply_attrs), ())
#                            else:
#                                radiusd.radlog(radiusd.L_INFO, f"NTLM auth failed for {username_no_suffix}")
#                                return (radiusd.RLM_MODULE_REJECT, tuple(reply_attrs), ())
#                        else:
#                            # Not yet an MS-CHAP challenge, let FreeRADIUS continue
#                            radiusd.radlog(radiusd.L_INFO, "MSCHAP Challenge and Response Not Found.")
#                            return (radiusd.RLM_MODULE_OK, tuple(reply_attrs), ())
                    else:
                        # Outer EAP - do nothing, just OK
                        radiusd.radlog(radiusd.L_
            return (radiusd.RLM_MODULE_UPDATED, tuple(reply_attrs), tuple(control_attrs))

    else:
        radiusd.radlog(radiusd.L_INFO, "No matched policy, rejecting...")
        return (radiusd.RLM_MODULE_REJECT, (), ())

def accounting(p):
    # Convert received attributes
    attrs = dict(p)

    # You can print them to logs
    radiusd.radlog(radiusd.L_INFO, f"Accounting packet received: {attrs}")

    # Write them to a text file
    with open("/var/log/freeradius/accounting-events.log", "a") as f:
        f.write("New Accounting Record:\n")
        for k, v in attrs.items():
            f.write(f"  {k}: {v}\n")
        f.write("\n")

    # Always return OK (or you will get an error)
    return radiusd.RLM_MODULE_OK

#def post_auth(p):
#    eap_message = p.get("EAP-Message")
#
#    if not eap_message:
#        radiusd.radlog(radiusd.L_ERR, "No EAP-Message found.")
#        return radiusd.RLM_MODULE_NOOP
#
#    # Convert EAP-Message to bytes
#    eap_bytes = bytes.fromhex(eap_message.replace(" ", ""))
#
#    challenge_hex = parse_server_challenge(eap_bytes)
#    radiusd.radlog(radiusd.L_INFO, f"Parsed Server Challenge: {challenge_hex}")
#
#    # Store it keyed by State
#    CHALLENGE_CACHE["server_challenge"] = challenge_hex
#
```

```
157 #    radiusd.radlog(radiusd.L_INFO, f"Cached server challenge for state {state}:
    {challenge_hex}")
158 #    return radiusd.RLM_MODULE_OK
```

---

6. Supporting Modules

policy_matcher.py

- Encapsulates policy matching logic
- Allows complex expressions:
  - Time ranges
  - EPP status
  - Blacklist
  - RADIUS attributes
- **Extensible matching syntax (planned)**

authenticate.py

- Encapsulates authentication logic
- Routes authentication method based on policy:
  - SQL
  - YAML
  - LDAP
- **Extensible by any means of authentication logic (planned)**

reply_actions.py

- Responsible for:
  - Generating VLAN replies
  - RADIUS attributes
  - Custom messages
  - Applying logic from `dynamic_authorization.py`
- **Extensible action mapping**

dynamic_authorization.py

- Encodes logic for:
  - VLAN assignments for specific authorization
- **Extensible with other RADIUS attributes**

Authentication Modules

These modules implement backend-specific authentication:

1. auth_ldap.py

- LDAP (e.g., Active Directory)

Implemented Proof of Concept

```
import ldap3
import radiusd

def authenticate_ldap(auth_config, username, password):
    """
    Attempts LDAP bind-as-user authentication.
    Returns True if credentials are valid.
    """
    server = ldap3.Server(auth_config["server"], port=auth_config.get("port", 389),
get_info=ldap3.NONE)
    try:
        conn = ldap3.Connection(
            server,
            user=auth_config["bind_dn"],
            password=auth_config["bind_password"],
            auto_bind=True
        )
    except ldap3.LDAPBindError as e:
        radiusd.radlog(radiusd.L_ERR, f"LDAP bind error: {e}")
        return False

    user_filter = auth_config["user_filter"].replace("{username}", username)
    conn.search(auth_config["base_dn"], user_filter, attributes=["dn"])
    if not conn.entries:
        radiusd.radlog(radiusd.L_ERR, f"LDAP user {username} not found.")
        return False
    user_dn = conn.entries[0].entry_dn
    user_conn = ldap3.Connection(server, user=user_dn, password=password)
    if not user_conn.bind():
        radiusd.radlog(radiusd.L_ERR, f"LDAP bind failed for {username}:
{user_conn.result}")
        return False
    radiusd.radlog(radiusd.L_INFO, f"LDAP bind succeeded for {username}.")
    return True
```

2. auth_sql.py

- SQL authentication (MySQL, PostgreSQL)

Placeholder for future implementation

3. auth_oracle.py

- Oracle authentication

Placeholder for future implementation

4. auth_azure.py

- Azure AD authentication

Placeholder for future implementation

5. auth_custom_list.py

- Custom user list in the form of yaml

Placeholder for future implementation

Additional Extensibility Modules

1. blacklist.py

- Implements dynamic or static blacklists:
  - MAC addresses
  - Usernames
  - IP addresses

Placeholder

2. whitelist.py

- Implements pre-approved lists (VIP users, devices, mac oui db)
  - Acts as if authentication source is not selected

Placeholder

3. epp_inventory.py

- Integrates with endpoint inventory
  - Intended for `policy_matcher.py` logic (Extensible for authentication logic for limitations)

Placeholder

Configuration Files

- `policies.yaml` : Defines policies and actions
  - ```yaml
    policies:
      - name: corp_vlan
        match:
          nas_ip: "192.168.100.10"
        action:
          accept: true
          vlan: "100"
          reply_message: "Corporate Access Granted"
        auth_source: "ad_ldap"

      - name: guest_mac
        match:
          mac_prefix: "AA:BB:CC"
        action:
          accept: true
          vlan: "200"
          reply_message: "Guest Access Granted"

      - name: contractors
        match:
          username_suffix: "@contractor"
        action:
          accept: true
          vlan: "300"
          reply_message: "Contractor Access Granted"

    ```
    ```

```
  - name: user
    match:
      username: "ege"
    action:
      accept: true
      vlan: "400"
      reply_message: "User Access Granted"
    auth_source: "ad_ldap"

  - name: local_test
    match:
      username: "alice"
    action:
      accept: true
      vlan: "500"
      reply_message: "Local User Access Granted"
    auth_source: "local_user_list"

  - name: eap_tls_user
    match:
      username: "client"
    action:
      accept: true
      vlan: "600"
      reply_message: "EAP-TLS User Access"

  - name: default_deny
    match: {}
    action:
      reject: true
      reply_message: "Access Denied"
```

- `auth_sources.yaml` : Defines backend authentication credentials
  - ```
    auth_sources:
      ad_ldap:
        type: ldap
        server: 10.34.10.4
        port: 389
        bind_dn: "CN=Administrator,CN=Users,DC=fnss,DC=local"
        bind_password: "Deneme12"
        base_dn: "DC=fnss,DC=local"
        user_filter: "(sAMAccountName={username})"
        domain: "FNSS"

      ad_ldap_backup:
        type: ldap
        server: 10.34.10.5
        port: 389
        bind_dn: "CN=radius_bind,OU=ServiceAccounts,DC=example,DC=com"
        bind_password: "..."
        base_dn: "DC=example,DC=com"
        user_filter: "(sAMAccountName={username})"
        domain: "EXAMPLE"

      local_user_list:
        type: yaml
        path: /etc/freeradius/3.0/mods-config/python3/local_credentials.yaml

    ```
    ```

```
  local_db:
    type: sql
    server: 127.0.0.1
    db: "va"
    db_user: "vadmin"
    db_password: "991550sE*"
    schema: "public"
    table: "radcheck"
    user_column: "username"
    password_column: "password"
```

- `.reload_policies` : Reload trigger file

Accounting

Description:

- Accounting requests are sent by NAS to log start, interim, and stop events.
- Configured `listen` block on port 1813:

```
listen {
    ipaddr = *
    port = 1813
    type = acct
}
```

- Python `accounting()` hook captures all accounting packets and should parse and write to a database table.

FreeRADIUS Virtual Server

A single FreeRADIUS virtual server handles all the communication whether inner or outer tunnel of protocols by looping in on itself. Utilizes FreeRADIUS modules for decoding protocols and hooking into external python logic whenever a decision is to made.

```
server s3m_all {
    listen {
        ipaddr = *
        port = 1812
        type = auth
    }

    listen {
        ipaddr = *
        port = 1813
        type = acct
    }

    authorize {
        eap
        python3
        mschap
    }

    authenticate {
        Auth-Type EAP {
            eap
```

```
        }
        Auth-Type PAP {
            pap
        }
        Auth-Type MS-CHAP {
            if (&control:Cleartext-Password) {
                mschap_local
            } else {
                mschap
            }
        }
    }

    post-auth {
    }

    accounting {
        python3
    }
}
```

FreeRADIUS Dynamic Clients

/etc/freeradius/3.0/mods-available/sql

```
sql local{
    driver = "rlm_sql_postgresql"
    dialect = "postgresql"

    server = "localhost"
    port = 5432
    login = "vadmin"
    password = "991550sE*"

    radius_db = "va"
    read_clients = yes

    client_table = "public.nas"
}
```

Activate:

```
ln -s /etc/freeradius/3.0/mods-available/sql /etc/freeradius/3.0/mods-enabled
```

Future Considerations

- **Policy Versioning:** Include version stamps and migration tooling
- **Validation:** Pre-flight checks for YAML correctness before reload
- **Audit Logging:** Record authentication and policy matches
- **UI Integration:** Web console for policy editing
- **Dynamic Updates:** Push configuration changes via API without touching files

---

Developer Workflow

When adding new modules:

1. Create `<module>.py` in `mods-config/python3/`
2. Implement a `def authenticate(config, username, password):` or equivalent function in dispatcher.py
3. Extend `policies.yaml` and `auth_sources.yaml` schema to reference the module
4. Test with `radtest` and `eapol_test`

Next Steps for Developers

1. Create each mentioned extension module
2. Abstract use of each module to configuration level from dispatcher.py
3. Write a compiler for transition of data from database to yaml files
4. Extensive logging per procedure and state

Technical Details

PAP Authentication

Description:

- The simplest form—username and password in clear text.
- Handled by the `pap` module.

Key points:

- Easy to integrate with multiple authentication sources.
- Suitable for environments where EAP is not required.

---

2. MS-CHAPv2

Description:

- Commonly used for Windows native 802.1X supplicants.
- Supports both:
  - **Winbind + ntlm_auth** (Active Directory)
  - **Cleartext Password** (local validation)

Implementation Details:

- `mschap` module configured with:

```
ntlm_auth = "/usr/bin/ntlm_auth --request-nt-key --allow-mschapv2 --username=%
{mschap:User-Name} --challenge=%{mschap:Challenge} --nt-response=%{mschap:NT-Response}"
```

- **Dynamic switching logic:**

```
Auth-Type MS-CHAP {
```

```
    if (&control:Cleartext-Password) {
        mschap_local
    } else {
        mschap
    }
}
```

- `mschap_local` performs local password validation by hashing Cleartext-Password.
- Cleartext password sourced from `local_user_list.yaml`.

Notes:

- Allows per-policy selection of Active Directory vs. local authentication.
- Dependency of Winbind relies on server joining a domain which creates hard limitation on a single use of Active Directory source.

---

3. EAP-MSCHAPv2

Description:

- Encapsulates MS-CHAPv2 inside EAP tunnel.
- Typical in PEAP (Protected EAP) usage.
- Requires:
  - Inner authentication using `mschap`
  - TLS tunnel with server certificate
- The intricacy of this design utilizes **same virtual server** processes for both outer and inner tunnels.

Behavior:

- EAP decodes MSCHAPv2 challenge/response.
- Python module inspects the outer/inner identity and sets policy.
- Final authentication handled by `mschap` or `mschap_local` as above.

EAP-TTLS

Description:

- Creates a TLS tunnel, then performs inner authentication (e.g., PAP or MSCHAPv2).
- Supports:
  - EAP-TTLS/PAP
  - EAP-TTLS/MSCHAPv2
- Flexibility for different client capabilities.

Implementation:

- No additional FreeRADIUS modules required beyond `eap` and `mschap`.
- Policies remain centralized in the same Python module.

EAP-TLS

Description:

- Certificate-based mutual authentication.
- No username/password—identity derived from client certificate.

Proof of Concept

Freeradius Installation

Package installation (Ubuntu):

```
apt update
apt install freeradius freeradius-utils freeradius-ldap freeradius-python3 samba winbind
ntpdate ldap-utils wpasupplicant
```

Winbind + Active Directory

Configure smb.conf

Path:

```
/etc/samba/smb.conf
```

Example minimal config:

```
[global]
   workgroup = FNSS
   security = ADS
   realm = FNSS.LOCAL
   encrypt passwords = yes
   idmap config * :              backend = tdb
   idmap config * :              range   = 3000-7999
   idmap config FNSS : backend = rid
   idmap config FNSS : range   = 100000-999999
   winbind use default domain = yes
   winbind offline logon = yes
```

Sync NTP:

```
# After setting write timedatectl zone
timedatectl set-timezone Europe/Istanbul
ntpdate 10.34.10.4
```

Test domain join:

```
net ads testjoin
wbinfo -u
```

`ntlm_auth` test:

```
ntlm_auth --request-nt-key --domain=FNSS --username=ege --password=YourPassword
```

Certificates for EAP-TLS / PEAP

Generate built-in FreeRADIUS test certificates:

```
cd /etc/freeradius/3.0/certs
make
```

Creates:

- `server.pem` (certificate)
- `server.key` (private key)
- `ca.pem` (CA)

Referenced in `mods-enabled/eap`:

```
tls-config tls-common {
   private_key_password = whatever
   private_key_file = /etc/freeradius/3.0/certs/server.key
   certificate_file = /etc/freeradius/3.0/certs/server.pem
   ca_file = /etc/freeradius/3.0/certs/ca.pem
   random_file = /dev/urandom
   require_client_cert = yes
}
```

EAP

mods-enabled/eap

- Configured for:
  - PEAP
  - EAP-TTLS
  - EAP-TLS

Main section example:

```
eap {
   default_eap_type = md5
   timer_expire = 60
   ignore_unknown_eap_types = no
   cisco_accounting_username_bug = no

   tls-config tls-common {
       private_key_password = whatever
       private_key_file = /etc/freeradius/3.0/certs/server.key
       certificate_file = /etc/freeradius/3.0/certs/server.pem
       ca_file = /etc/freeradius/3.0/certs/ca.pem
       random_file = /dev/urandom
       require_client_cert = yes
   }

   ttls {
       virtual_server = "s3m_all"
   }

```

```
   peap {
       default_eap_type = mschapv2
       virtual_server = "s3m_all"
   }
}
```

MS-CHAP

mods-enabled/mschap

Default, with ntlm_auth:

```
mschap {
   use_mppe = yes
   require_encryption = yes
   require_strong = yes
   with_ntdomain_hack = yes
   ntlm_auth = "/usr/bin/ntlm_auth --request-nt-key --allow-mschapv2 --username=%{%{Stripped-User-Name}:-%{%{User-Name}:-None}} --challenge=%{%{mschap:Challenge}:-00} --nt-response=%{%{mschap:NT-Response}:-00}"
}
```

MS-CHAP-LOCAL

mods-enabled/mschap_local

Without ntlm_auth:

```
mschap mschap_local {
   use_mppe = yes
   require_encryption = yes
   require_strong = yes
   with_ntdomain_hack = yes
#   ntlm_auth = "/usr/bin/ntlm_auth --request-nt-key --allow-mschapv2 --username=%{%{Stripped-User-Name}:-%{%{User-Name}:-None}} --challenge=%{%{mschap:Challenge}:-00} --nt-response=%{%{mschap:NT-Response}:-00}"
}
```

Python Module

mods-enabled/python3

```
python3 {
    python_path = "/etc/freeradius/3.0/mods-config/python3"
    module = dispatcher

    func_instantiate = instantiate
    mod_instantiate = ${.module}

    func_authorize = authorize
    mod_authorize = ${.module}

    func_accounting = accounting
    mod_accounting = ${.module}
}
```

Authentication Sources

Example YAML:

/etc/freeradius/3.0/mods-config/python3/auth_sources.yaml

```
auth_sources:
  ad_ldap:
    type: ldap
    server: 10.34.10.4
    port: 389
    bind_dn: "CN=Administrator,CN=Users,DC=fnss,DC=local"
    bind_password: "Deneme12"
    base_dn: "DC=fnss,DC=local"
    user_filter: "(sAMAccountName={username})"
    domain: "FNSS"

  ad_ldap_backup:
    type: ldap
    server: 10.34.10.5
    port: 389
    bind_dn: "CN=radius_bind,OU=ServiceAccounts,DC=example,DC=com"
    bind_password: "..."
    base_dn: "DC=example,DC=com"
    user_filter: "(sAMAccountName={username})"
    domain: "EXAMPLE"

  local_user_list:
    type: yaml
    path: /etc/freeradius/3.0/mods-config/python3/local_credentials.yaml

  local_db:
    type: sql
    server: 127.0.0.1
    db: "va"
    db_user: "vadmin"
    db_password: "991550sE*"
    schema: "public"
    table: "radcheck"
    user_column: "username"
    password_column: "password"
```

LDAP Authentication Logic

/etc/freeradius/3.0/mods-config/python3/auth_ldap.py

```
import ldap3
import radiusd

def authenticate_ldap(auth_config, username, password):
    """
    Attempts LDAP bind-as-user authentication.
    Returns True if credentials are valid.
    """
    server = ldap3.Server(auth_config["server"], port=auth_config.get("port", 389),
get_info=ldap3.NONE)
```

```
    try:
        conn = ldap3.Connection(
            server,
            user=auth_config["bind_dn"],
            password=auth_config["bind_password"],
            auto_bind=True
        )
    except ldap3.LDAPBindError as e:
        radiusd.radlog(radiusd.L_ERR, f"LDAP bind error: {e}")
        return False

    user_filter = auth_config["user_filter"].replace("{username}", username)
    conn.search(auth_config["base_dn"], user_filter, attributes=["dn"])
    if not conn.entries:
        radiusd.radlog(radiusd.L_ERR, f"LDAP user {username} not found.")
        return False
    user_dn = conn.entries[0].entry_dn
    user_conn = ldap3.Connection(server, user=user_dn, password=password)
    if not user_conn.bind():
        radiusd.radlog(radiusd.L_ERR, f"LDAP bind failed for {username}:
{user_conn.result}")
        return False
    radiusd.radlog(radiusd.L_INFO, f"LDAP bind succeeded for {username}.")
    return True

```

Local User Credentials

Example YAML:

/etc/freeradius/3.0/mods-config/python3/local_credentials.yaml

```
credentials:
  ege: Deneme12
  alice: s3cr3t
  bob: password123
```

Policy Definitions

Example YAML:

/etc/freeradius/3.0/mods-config/python3/policies.yaml

```
policies:
  - name: corp_vlan
    match:
      nas_ip: "192.168.100.10"
    action:
      accept: true
      vlan: "100"
      reply_message: "Corporate Access Granted"
    auth_source: "ad_ldap"

  - name: guest_mac
    match:
      mac_prefix: "AA:BB:CC"
```

```
    action:
      accept: true
      vlan: "200"
      reply_message: "Guest Access Granted"

  - name: contractors
    match:
      username_suffix: "@contractor"
    action:
      accept: true
      vlan: "300"
      reply_message: "Contractor Access Granted"

  - name: user
    match:
      username: "ege"
    action:
      accept: true
      vlan: "400"
      reply_message: "User Access Granted"
    auth_source: "ad_ldap"

  - name: local_test
    match:
      username: "alice"
    action:
      accept: true
      vlan: "500"
      reply_message: "Local User Access Granted"
    auth_source: "local_user_list"

  - name: eap_tls_user
    match:
      username: "client"
    action:
      accept: true
      vlan: "600"
      reply_message: "EAP-TLS User Access"

  - name: default_deny
    match: {}
    action:
      reject: true
      reply_message: "Access Denied"
```

Accounting Log File

/var/log/freeradius_accounting.log

```
touch /var/log/freeradius/accounting-events.log
chown freerad:freerad /var/log/freeradius/accounting-events.log
```

PAP (radclient)

```
echo "User-Name = ege@fnss.local, User-Password = Deneme12, Calling-Station-Id = AA:BB:CC:DD:EE:FF, Nas-Ip-Address = 192.168.100.10" | radclient -x 127.0.0.1 auth testing123
```

MSCHAPv2 (radtest)

```
radtest -t mschap ege Deneme12 localhost 0 testing123
```

EAP-PEAP / EAP-MSCHAPv2 (eapol)

/root/eaptest/eapol_eap_mschapv2.conf

```
network={
    key_mgmt=WPA-EAP
    eap=PEAP
    identity="ege"
    password="Deneme12"
    phase2="auth=MSCHAPV2"
}
```

Run:

```
eapol_test -c eapol_eap_mschapv2.conf -a 127.0.0.1 -s testing123
```

EAP-TTLS/PAP (eapol)

/root/eaptest/eapol_ttls_pap.conf

```
network={
    ssid="test"
    key_mgmt=WPA-EAP
    eap=TTLS
    identity="ege"
    password="Deneme12"
    phase2="auth=PAP"
}
```

Run:

```
eapol_test -c eapol_ttls_pap.conf -a 127.0.0.1 -s testing123
```

EAP-TTLS/MSCHAPv2 (eapol)

/root/eaptest/eapol_ttls_mschapv2.conf

```
network={
    ssid="test"
    key_mgmt=WPA-EAP
    eap=TTLS
    identity="ege"
    password="Deneme12"
    phase2="auth=MSCHAPV2"
```

```
}
```

Run:

```
eapol_test -c eapol_ttls_mschapv2.conf -a 127.0.0.1 -s testing123
```

EAP-TLS (eapol)

/root/eaptest/eapol_tls.conf

```
network={
    key_mgmt=WPA-EAP
    eap=TLS
    identity="client"
    ca_cert="/etc/freeradius/3.0/certs/ca.pem"
    client_cert="/etc/freeradius/3.0/certs/client.pem"
    private_key="/etc/freeradius/3.0/certs/client.key"
    private_key_passwd="whatever"
}
```

Run:

```
eapol_test -c eapol_tls.conf -a 127.0.0.1 -s testing123
```

Accounting Requests

Start:

```
echo "
User-Name = 'ege'
Acct-Status-Type = Start
NAS-IP-Address = 192.168.1.1
Acct-Session-Id = '123456'
Framed-IP-Address = 10.0.0.50
NAS-Port = 0
" | radclient -x 127.0.0.1:1813 acct testing123
```

Interim:

```
echo "
User-Name = 'ege'
Acct-Status-Type = Interim-Update
NAS-IP-Address = 192.168.1.1
Acct-Session-Id = '123456'
Acct-Input-Octets = 500000
Acct-Output-Octets = 200000
" | radclient -x 127.0.0.1:1813 acct testing123
```

Stop:

```
echo "
User-Name = 'ege'
Acct-Status-Type = Stop
NAS-IP-Address = 192.168.1.1
```

```
Acct-Session-Id = '123456'
Acct-Session-Time = 3600
Acct-Input-Octets = 1000000
Acct-Output-Octets = 500000
" | radclient -x 127.0.0.1:1813 acct testing123
```

Future Design Improvement

Network Segmentation and Multi-Tenant Readiness

Principle:

- Multiple `listen` blocks per NIC/IP/port to segment incoming traffic.
- All blocks can route to:
  - A **single virtual server** (shared policies and logic)
  - Or **distinct virtual servers** (custom modules and certificates per tenant)
- For ultimate separation, **Docker containers** can be used per tenant:
  - Each container has:
    - Its own FreeRADIUS instance
    - Dedicated `winbind` domain membership
    - Separate certificates
    - Isolated configurations