FreeRADIUS & rlm_python

What is `rlm_python`?

It's a FreeRADIUS module that lets you:

- Intercept `authorize`, `authenticate`, `post-auth`, `accounting`, etc.
- Execute arbitrary Python logic during these phases
- Read request attributes and modify reply or control attributes
- Dynamically pull data from a DB or API

1. Install `rlm_python`

On Debian/Ubuntu:

```
sudo apt install freeradius-python3
```

2. Enable the Module

```
cd /etc/freeradius/3.0/mods-enabled/
ln -s ../mods-available/python python
```

3. Configure the Module

Edit:
`/etc/freeradius/3.0/mods-available/python`

Set:

```
python {
    module = policy_engine
    python_path = ${modconfdir}/${.:name}/site-packages
}
```

Then create the directory:

```
mkdir -p /etc/freeradius/3.0/mods-config/python/site-packages/
```

4. Python File (policy_engine.py)

Create:
`/etc/freeradius/3.0/mods-config/python/site-packages/policy_engine.py`

Example skeleton:

```
def authorize(p):
    username = p["User-Name"]
    nas_ip = p.get("NAS-IP-Address", "unknown")

    # Load policies from DB (you can use sqlalchemy or psycopg2 here)
    policy = match_policy(username, nas_ip, p)

    if not policy:
```

```python
        return 1  # Reject

    backend = get_backend(policy['auth_backend'])

    success, user_data = backend.authenticate(username, p["User-Password"])
    if not success:
        return 1  # Reject

    if policy.get("check_blacklist", False):
        if is_blacklisted(username, user_data["mac"]):
            return 1  # Reject

    # VLAN reply attribute example
    p["reply:Tunnel-Type"] = "VLAN"
    p["reply:Tunnel-Medium-Type"] = "IEEE-802"
    p["reply:Tunnel-Private-Group-Id"] = str(user_data["vlan_id"])

    return 2  # OK

def match_policy(username, nas_ip, p):
    # Query policy DB table, match conditions
    # Return best-matching policy dict
    return {
        "auth_backend": "AD",
        "check_blacklist": True
    }

def get_backend(backend_type):
    if backend_type == "AD":
        return ADBackend()
    elif backend_type == "TLS":
        return TLSBackend()
    elif backend_type == "ORACLE":
        return OracleBackend()
    else:
        raise Exception("Unknown backend")

class ADBackend:
    def authenticate(self, username, password):
        # Do LDAP bind etc.
        return True, {"vlan_id": 100, "mac": "00:11:22:33:44:55"}

# Add TLSBackend, OracleBackend, etc.

```

5. Call It in `sites-enabled/default`

Inside the `authorize` section:

```
authorize {
    ...
    python
    ...
}
```

You can also use it in:

- `authenticate`

- `post-auth`
- `accounting`

Advanced Enhancements

Blacklist Grouping

Make the blacklist a table with:

- `id`
- `mac`, `username`, etc.
- `group_id`
- `reason`
- `expires_at`

Each policy can define:

- `blacklist_scope = "group"` or `"global"`

Python logic filters based on this.

Failover Backends

If one backend (e.g., AD) fails, allow fallback to a secondary (e.g., local DB), configured per policy. Easy to express in a `backends: [AD, LOCAL]` list and iterate.

Logging & Auditing

Log every auth attempt with:

- Matched policy ID
- User info
- Result (success/fail)
- Reason (blacklist, bad password, etc.)

Use syslog, flat file, or DB insert.

Python Implementation (Dynamic Policy Matching)

```
import psycopg2

def match_policy(username, nas_ip, request_attrs):
    conn = psycopg2.connect(...)  # Your DB connection
    cur = conn.cursor()

    # Get all services
    cur.execute("SELECT id, name, auth_backend, check_blacklist FROM rad_services")
    services = cur.fetchall()

    for service in services:
        service_id, name, backend, check_bl = service
        if service_matches(service_id, request_attrs, cur):
            return {
                "id": service_id,
                "name": name,
                "auth_backend": backend,
```

```python
                "check_blacklist": check_bl
            }

    return None

def service_matches(service_id, attrs, cur):
    cur.execute("""
        SELECT attr_name, operator, attr_value
        FROM rad_services_control
        WHERE rad_service_id = %s
    """, (service_id,))

    conditions = cur.fetchall()

    for attr_name, op, val in conditions:
        req_val = attrs.get(attr_name)
        if req_val is None:
            return False

        # Convert both to strings for simplicity
        req_val = str(req_val)
        val = str(val)

        if not eval_condition(req_val, op, val):
            return False

    return True

def eval_condition(value, operator, expected):
    if operator == "==":
        return value == expected
    elif operator == "!=":
        return value != expected
    elif operator == "startswith":
        return value.startswith(expected)
    elif operator == "endswith":
        return value.endswith(expected)
    elif operator == "contains":
        return expected in value
    elif operator == "in":
        # expected = "val1,val2,val3"
        return value in expected.split(",")
    else:
        return False
```

Integration With FreeRADIUS Python Hook

```python
def authorize(p):
    username = p.get("User-Name")
    nas_ip = p.get("NAS-IP-Address")

    # Convert request to flat dict of radius attributes
    request_attrs = {k: str(v) for k, v in p.items()}

    policy = match_policy(username, nas_ip, request_attrs)
    if not policy:
        return 1  # Reject

    backend = get_backend(policy["auth_backend"])
    success, user_data = backend.authenticate(username, p["User-Password"])

    if not success:
        return 1

    if policy["check_blacklist"]:
        if is_blacklisted(username, user_data["mac"]):
            return 1

    # Example reply
    p["reply:Tunnel-Type"] = "VLAN"
    p["reply:Tunnel-Medium-Type"] = "IEEE-802"
    p["reply:Tunnel-Private-Group-Id"] = str(user_data["vlan_id"])

    return 2
```

- For performance, you could **cache** `rad_services` and `rad_services_control` in memory or Redis and refresh every minute.

Python Logic to Add Replies Dynamically

Extend the `authorize()` function in the Python module:

```python
def set_reply_attributes(service_id, p, cur):
    cur.execute("""
        SELECT attr_name, attr_value
        FROM rad_service_replies
        WHERE rad_service_id = %s
    """, (service_id,))

    for attr_name, attr_value in cur.fetchall():
        # Explicitly set reply attributes
        p[f"reply:{attr_name}"] = attr_value
```

Full integration:

```python
def authorize(p):
    username = p.get("User-Name")
    nas_ip = p.get("NAS-IP-Address")

    request_attrs = {k: str(v) for k, v in p.items()}

    conn = psycopg2.connect(...)  # reuse connection pool ideally
    cur = conn.cursor()

    policy = match_policy(username, nas_ip, request_attrs, cur)
    if not policy:
        return 1  # Reject

    backend = get_backend(policy["auth_backend"])
    success, user_data = backend.authenticate(username, p["User-Password"])

    if not success:
        return 1

    if policy["check_blacklist"]:
        if is_blacklisted(username, user_data["mac"]):
            return 1

    # Inject dynamic replies
    set_reply_attributes(policy["id"], p, cur)

    return 2  # OK
```

And use Python to evaluate conditions like:

```
def conditional_attr_match(attr_val, op, expected):
    if op == "==":
        return attr_val == expected
    elif op == "startswith":
        return attr_val.startswith(expected)
    elif op == "!=":
        return attr_val != expected
    return False
```