ASSIGNMENT – 20

TASK 1 :

```python
import os

def connect_to_database(host, port, db_name, user, password):
    """
    Connects to a generic database.

    Args:
        host: Database host.
        port: Database port.
        db_name: Database name.
        user: Username for authentication.
        password: Password for authentication.

    Returns:
        A connection object (placeholder).
    """
    print(f"Attempting to connect to database:")
    print(f"  Host: {host}")
    print(f"  Port: {port}")
    print(f"  Database: {db_name}")
    print(f"  User: {user}")
    print(f"  Password: {'*' * len(password)}") # Mask the password

    # In a real scenario, you would use a library like psycopg2, mysql.connector, etc.
    # For demonstration, we'll just return a placeholder.
    connection = f"Database Connection Object for {db_name}"
    return connection

# Example usage with placeholder values
db_host = 'DATABASE_HOST_PLACEHOLDER'
db_port = 'DATABASE_PORT_PLACEHOLDER'
db_name = 'DATABASE_NAME_PLACEHOLDER'
db_user = 'DATABASE_USER_PLACEHOLDER'
db_password = 'DATABASE_PASSWORD_PLACEHOLDER'

# You would typically load these from environment variables or a config file
# For this subtask, we use placeholders directly.

db_connection = connect_to_database(db_host, db_port, db_name, db_user, db_password)
print(f"Connection Status: {db_connection}")

def connect_to_api(api_endpoint, api_key):
    """
    Connects to a generic API.

    Args:
        api_endpoint: API endpoint URL.
        api_key: API key for authentication.

    Returns:
        An API client object (placeholder).
    """
    print(f"\nAttempting to connect to API:")
    print(f"  Endpoint: {api_endpoint}")
    print(f"  API Key: {'*' * len(api_key)}") # Mask the API key

    # In a real scenario, you would use a library like requests
    # For demonstration, we'll just return a placeholder.
    api_client = f"API Client Object for {api_endpoint}"
    return api_client

# Example usage with placeholder values
api_url = 'API_ENDPOINT_PLACEHOLDER'
api_secret_key = 'API_KEY_PLACEHOLDER'

# You would typically load these from environment variables or a config file
# For this subtask, we use placeholders directly.

api_connection = connect_to_api(api_url, api_secret_key)
print(f"Connection Status: {api_connection}")
```

```
Attempting to connect to database:
  Host: DATABASE_HOST_PLACEHOLDER
  Port: DATABASE_PORT_PLACEHOLDER
  Database: DATABASE_NAME_PLACEHOLDER
  User: DATABASE_USER_PLACEHOLDER
  Password: ********************************
Connection Status: Database Connection Object for DATABASE_NAME_PLACEHOLDER

Attempting to connect to API:
  Endpoint: API_ENDPOINT_PLACEHOLDER
  API Key: ********************
Connection Status: API Client Object for API_ENDPOINT_PLACEHOLDER
```

```
# The code from the previous step is:
# import os

# def connect_to_database(host, port, db_name, user, password):
#     """
#     Connects to a generic database.

#     Args:
#         host: Database host.
#         port: Database port.
#         db_name: Database name.
#         user: Username for authentication.
#         password: Password for authentication.

#     Returns:
#         A connection object (placeholder).
#     """
#     print(f"Attempting to connect to database:")
#     print(f"  Host: {host}")
#     print(f"  Port: {port}")
#     print(f"  Database: {db_name}")
#     print(f"  User: {user}")
#     print(f"  Password: {'*' * len(password)}") # Mask the password

#     # In a real scenario, you would use a library like psycopg2, mysql.connector, etc.
#     # For demonstration, we'll just return a placeholder.
#     connection = f"Database Connection Object for {db_name}"
#     return connection

# # Example usage with placeholder values
# db_host = 'DATABASE_HOST_PLACEHOLDER'
# db_port = 'DATABASE_PORT_PLACEHOLDER'
# db_name = 'DATABASE_NAME_PLACEHOLDER'
# db_user = 'DATABASE_USER_PLACEHOLDER'
# db_password = 'DATABASE_PASSWORD_PLACEHOLDER'

# # You would typically load these from environment variables or a config file
# # For this subtask, we use placeholders directly.

# db_connection = connect_to_database(db_host, db_port, db_name, db_user, db_password)
# print(f"Connection Status: {db_connection}")

# def connect_to_api(api_endpoint, api_key):
#     """
#     Connects to a generic API.

#     Args:
#         api_endpoint: API endpoint URL.
#         api_key: API key for authentication.

#     Returns:
#         An API client object (placeholder).
#     """
#     print(f"\nAttempting to connect to API:")
#     print(f"  Endpoint: {api_endpoint}")
#     print(f"  API Key: {'*' * len(api_key)}") # Mask the API key

#     # In a real scenario, you would use a library like requests
#     # For demonstration, we'll just return a placeholder.
#     api_client = f"API Client Object for {api_endpoint}"
#     return api_client

# # Example usage with placeholder values
# api_url = 'API_ENDPOINT_PLACEHOLDER'
# api_secret_key = 'API_KEY_PLACEHOLDER'

# # You would typically load these from environment variables or a config file
# # For this subtask, we use placeholders directly.

# api_connection = connect_to_api(api_url, api_secret_key)
# print(f"Connection Status: {api_connection}")

# Based on the review, the following lines contain placeholder values, not hardcoded credentials:
# db_host = 'DATABASE_HOST_PLACEHOLDER'
# db_port = 'DATABASE_PORT_PLACEHOLDER'
# db_name = 'DATABASE_NAME_PLACEHOLDER'
# db_user = 'DATABASE_USER_PLACEHOLDER'
# db_password = 'DATABASE_PASSWORD_PLACEHOLDER'
# api_url = 'API_ENDPOINT_PLACEHOLDER'
# api_secret_key = 'API_KEY_PLACEHOLDER'

# No hardcoded usernames, passwords, or API keys were found in the generated code.
```

```python
import os

def connect_to_database(host, port, db_name, user, password):
    """
    Connects to a generic database.

    Args:
        host: Database host.
        port: Database port.
        db_name: Database name.
        user: Username for authentication.
        password: Password for authentication.

    Returns:
        A connection object (placeholder).
    """
    print(f"Attempting to connect to database:")
    print(f"  Host: {host}")
    print(f"  Port: {port}")
    print(f"  Database: {db_name}")
    print(f"  User: {user}")
    # Mask the password if it's not None
    password_display = '*' * len(password) if password else 'None'
    print(f"  Password: {password_display}")

    # In a real scenario, you would use a library like psycopg2, mysql.connector, etc.
    # For demonstration, we'll just return a placeholder.
    connection = f"Database Connection Object for {db_name}"
    return connection

# Load database credentials from environment variables
db_host = os.environ.get('DATABASE_HOST')
db_port = os.environ.get('DATABASE_PORT')
db_name = os.environ.get('DATABASE_NAME')
db_user = os.environ.get('DATABASE_USER')
db_password = os.environ.get('DATABASE_PASSWORD')

db_connection = connect_to_database(db_host, db_port, db_name, db_user, db_password)
print(f"Connection Status: {db_connection}")

def connect_to_api(api_endpoint, api_key):
    """
    Connects to a generic API.

    Args:
        api_endpoint: API endpoint URL.
        api_key: API key for authentication.

    Returns:
        An API client object (placeholder).
    """
    print(f"\nAttempting to connect to API:")
    print(f"  Endpoint: {api_endpoint}")
    # Mask the API key if it's not None
    api_key_display = '*' * len(api_key) if api_key else 'None'
    print(f"  API Key: {api_key_display}")

    # In a real scenario, you would use a library like requests
    # For demonstration, we'll just return a placeholder.
    api_client = f"API Client Object for {api_endpoint}"
    return api_client

# Load API credentials from environment variables
api_url = os.environ.get('API_ENDPOINT')
api_secret_key = os.environ.get('API_KEY')

api_connection = connect_to_api(api_url, api_secret_key)
print(f"Connection Status: {api_connection}")
```

```
Attempting to connect to database:
  Host: None
  Port: None
  Database: None
  User: None
  Password: None
Connection Status: Database Connection Object for None

Attempting to connect to API:
  Endpoint: None
  API Key: None
Connection Status: API Client Object for None
```

TASK 2 :

```python
import sqlite3

def create_database():
    conn = None
    try:
        conn = sqlite3.connect('users.db')
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                username TEXT PRIMARY KEY,
                password TEXT
            )
        ''')
        conn.commit()
        print("Database and table created successfully.")
    except sqlite3.Error as e:
        print(f"Error creating database: {e}")
    finally:
        if conn:
            conn.close()

def register_user(username, password):
    conn = None
    try:
        conn = sqlite3.connect('users.db')
        cursor = conn.cursor()
        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, password))
        conn.commit()
        print(f"User '{username}' registered successfully.")
    except sqlite3.IntegrityError:
        print(f"Error: Username '{username}' already exists.")
    except sqlite3.Error as e:
        print(f"Error registering user: {e}")
    finally:
        if conn:
            conn.close()

def login_user(username, password):
    conn = None
    try:
        conn = sqlite3.connect('users.db')
        cursor = conn.cursor()
        # WARNING: Vulnerable to SQL injection
        query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
        cursor.execute(query)
        user = cursor.fetchone()
        if user:
            print(f"Login successful for user '{username}'.")
            return True
        else:
            print(f"Login failed for user '{username}'.")
            return False
    except sqlite3.Error as e:
        print(f"Error during login: {e}")
        return False
    finally:
        if conn:
            conn.close()

# Initialize the database
create_database()

# Register a test user
register_user("testuser", "password123")
```

```
Database and table created successfully.
User 'testuser' registered successfully.
```

TASK 3 :

```python
import sqlite3

def secure_login(username, password):
    """Checks if the provided username and password match a user in the database
    using parameterized queries to prevent SQL injection.
    """
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    query = "SELECT * FROM users WHERE username = ? AND password = ?"
    cursor.execute(query, (username, password))
    user = cursor.fetchone()
    conn.close()
    return user is not None
```

```python
injection_username = "' OR '1'='1"
injection_password = "" # Any password will work with the injection
login_successful = login(injection_username, injection_password)
print(f"Login attempt with SQL injection: {login_successful}")
```

Identify vulnerabilities

```python
import sqlite3

def create_database():
    """Connects to a SQLite database and creates a users table with a sample user."""
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            username TEXT UNIQUE,
            password TEXT
        )
    ''')
    cursor.execute("INSERT OR IGNORE INTO users (username, password) VALUES (?, ?)", ('test_user', 'test_password'))
    conn.commit()
    conn.close()

def login(username, password):
    """Checks if the provided username and password match a user in the database."""
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)
    user = cursor.fetchone()
    conn.close()
    return user is not None

create_database()
```

TASK 4 :

```python
def unsafe_calculator():
  user_input = input("Enter a mathematical expression: ")
  try:
    # Vulnerable code using eval()
    result = eval(user_input)
    print(f"Result: {result}")
  except Exception as e:
    print(f"Error: {e}")

unsafe_calculator()
```

```
Enter a mathematical expression: 2+2
Result: 4
```

Now, here is a safer version of the calculator program that uses `ast.literal_eval()`. This function can safely evaluate strings containing Python literals (strings, numbers, tuples, lists, dicts, booleans, and None) but not arbitrary expressions.

```python
import ast

def safe_calculator():
  user_input = input("Enter a mathematical expression: ")
  try:
    # Secure code using ast.literal_eval()
    result = ast.literal_eval(user_input)
    print(f"Result: {result}")
  except (ValueError, SyntaxError) as e:
    print(f"Invalid input: {e}")
  except Exception as e:
    print(f"An unexpected error occurred: {e}")

safe_calculator()
```

```
Enter a mathematical expression: 2
Result: 2
```

TASK 5 :

🧪 Step 1: Insecure Flask Web Server (Initial Version)

Python Copy

```python
# app.py

from flask import Flask, request
import os

app = Flask(__name__)

@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form.get("cmd")
    os.system(cmd)  # 🚨 Vulnerability: Shell injection risk

    return "Command executed"

@app.route("/debug")
def debug():
    return eval("2 + 2")  # 🚨 Vulnerability: Use of eval


if __name__ == "__main__":
    app.run(debug=True)  # 🚨 Vulnerability: Debug mode exposes internals
```

🛡️ Step 2: Run Bandit Static Analysis

Bash Copy

```bash
bandit app.py
```

🔍 Bandit Output (Summary)

Copy

```
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call with
   Severity: High   Confidence: High
   Location: app.py:10

>> Issue: [B307:eval_used] Use of possibly insecure function - eval.
   Severity: Medium   Confidence: High
   Location: app.py:14

>> Issue: [B201:flask_debug_true] Flask app run with debug=True
   Severity: Medium   Confidence: High
   Location: app.py:17
```

✅ Step 3: Secure Flask Web Server (Fixed Version)

Python Copy

```python
# app.py

from flask import Flask, request
import subprocess

app = Flask(__name__)

@app.route("/run", methods=["POST"])
def run_command():
    cmd = request.form.get("cmd")
    try:
        # ✅ Use subprocess without shell=True

        result = subprocess.run(cmd.split(), capture_output=True, text=True,
        return result.stdout
    except Exception as e:
        return str(e), 400

@app.route("/debug")
def debug():
    return str(2 + 2)  # ✅ Removed eval


if __name__ == "__main__":
    app.run(debug=False)  # ✅ Turned off debug mode
```

🔁 Step 4: Re-run Bandit

Bash Copy

```
bandit app.py
```

✅ Bandit Output (Clean)

Copy

```
No issues identified.
```

✅ Bandit Scan Output (Secure Version)

Bash Copy

```
$ bandit app.py

[...]
--------------------------------------------------

Code scanned:
    Total lines of code: 20
    Total lines skipped (#nosec): 0

Run metrics:
    Total issues (by severity):
        High: 0
        Medium: 0
        Low: 0

No issues identified.
--------------------------------------------------
```