Input Validation Testing

3 min

Input validation is a process where you check if the data provided by a user or another system meets specific criteria before processing it. This typically involves ensuring that the data is of the expected type, within an acceptable range, formatted correctly, and does not contain malicious content.

Creating Input Validation Tests

When creating input validation tests, we want to systematically verify that an application's inputs correctly handle a wide range of data, including valid, invalid, and unexpected input. The goal is to ensure that the program behaves as expected and is robust against various input-related errors and security vulnerabilities.

What Will We Do?

In this lesson, we will use prompt engineering to aid in the creation of input validation tests. We will be supplied with Python code that does not have any unit tests. We must prompt the AI to create some tests to ensure that our code is free from security vulnerabilities. (*Note: You can copy the code into the chat to help get better results.*)

Instructions

Below is a Python script with known user input security vulnerabilities.

```
import re

def get_username():
    while True:
        username = input("Enter your username: ")
        # Simple validation to check for alphanumeric characters and underscores
        if re.match("^[a-zA-Z0-9_]*$", username):
            return username
        else:
            print("Invalid username. Only alphanumeric characters and underscores are allowed.")

def get_password():
    while True:
        password = input("Enter your password: ")
        # Password validation can be more complex, checking for length, complexity, etc.
        if len(password) >= 8:
            return password
        else:
            print("Invalid password. It must be at least 8 characters long.")

def get_email():
    while True:
        email = input("Enter your email address: ")
        # Simple regex pattern for validating an email address
        if re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$", email):
            return email
        else:
            print("Invalid email address format.")
```

```
def get_sql_query():
    while True:
        query = input("Enter your SQL query: ")
        # This is a trivial check; in a real scenario, use parameterized queries or ORM
        if "DROP" not in query.upper() and "DELETE" not in query.upper():
            return query
        else:
            print("Invalid query. Destructive operations are not allowed.")

# Usage example
def main():
    username = get_username()
    password = get_password()
    email = get_email()
    query = get_sql_query()

    print(f"Username: {username}")
    print(f"Password: {password}")
    print(f"Email: {email}")
    print(f"SQL Query: {query}")

if __name__ == "__main__":
    main()
```

Your task is to prompt the AI chat to develop unit tests for the Python script. Here's an example output from a prompt of a valid input validation test:

```
    def test_username_valid(self):
        self.assertEqual(get_username("valid_username123"), "valid_username123")
```

You will know you have successfully completed the task when you obtain the following results from the machine:

- Using the Python unittest framework
- get_username() should have at least three test cases: *a valid, invalid, and unexpected test*
- get_password() should have at least three test cases: *a valid, invalid, and unexpected test*
- get_email() should have at least three test cases: *a valid, invalid, and unexpected test*
- get_sql_query() should have at least three test cases: *a valid, invalid, and unexpected test*

Note: Depending on your prompt, you may receive different results than anticipated. That is okay. However, there should be a test for each function.

If you would like some help creating prompts to send to the AI, open the hint dropdown below.

Prompt Hint

The AI remembers previous messages so you can chat back and forth like a conversation. You may need to use multiple prompts, but combining prompts saves computation time and energy. Here are some examples below:

- Create unit tests for the following Python code using the unittest library: <paste in the code from above>
- Add test cases for valid, invalid, and unexpected data.

If you need the AI to forget past prompts, click the trash can icon and the chat will reset.

Prompt

Create unit tests for the following Python code using the unittest library:

```
import re

def get_username():
    while True:
        username = input("Enter your username: ")
        # Simple validation to check for alphanumeric characters and underscores
        if re.match("^[a-zA-Z0-9_]*$", username):
            return username
        else:
            print("Invalid username. Only alphanumeric characters and underscores are allowed.")

def get_password():
    while True:
        password = input("Enter your password: ")
        # Password validation can be more complex, checking for length, complexity, etc.
        if len(password) >= 8:
            return password
        else:
            print("Invalid password. It must be at least 8 characters long.")

def get_email():
    while True:
        email = input("Enter your email address: ")
        # Simple regex pattern for validating an email address
        if re.match(r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$", email):
            return email
        else:
            print("Invalid email address format.")

def get_sql_query():
    while True:
        query = input("Enter your SQL query: ")
        # This is a trivial check; in a real scenario, use parameterized queries or ORM
        if "DROP" not in query.upper() and "DELETE" not in query.upper():
            return query
        else:
```

```
        print("Invalid query. Destructive operations are not allowed.")

# Usage example
def main():
    username = get_username()
    password = get_password()
    email = get_email()
    query = get_sql_query()

    print(f"Username: {username}")
    print(f"Password: {password}")
    print(f"Email: {email}")
    print(f"SQL Query: {query}")

if __name__ == "__main__":
    main()
```

Response

```
import unittest

import re

from unittest.mock import patch

from io import StringIO


class TestUserInputValidation(unittest.TestCase):


    def test_get_username_valid(self):

        with patch('builtins.input', return_value="JohnDoe123"):

            self.assertEqual(get_username(), "JohnDoe123")


    def test_get_username_invalid(self):

        with patch('builtins.input', return_value="John_Doe"):

            with patch('sys.stdout', new_callable=StringIO) as mock_stdout:

                get_username()

                self.assertEqual(mock_stdout.getvalue().strip(), "Invalid username. Only alphanumeric
characters and underscores are allowed.")


    def test_get_username_unexpected(self):

        with patch('builtins.input', return_value="John@Doe"):

            self.assertIsNone(get_username())
    def test_get_password_valid(self):
        with patch('builtins.input', return_value="StrongPassword"):
            self.assertEqual(get_password(), "StrongPassword")

    def test_get_password_invalid(self):
        with patch('builtins.input', return_value="Weak"):
            with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
                get_password()
                self.assertEqual(mock_stdout.getvalue().strip(), "Invalid password. It must be at least 8 characters long.")

    def test_get_password_unexpected(self):
        with patch('builtins.input', return_value="Short"):
            self.assertIsNone(get_password())

    def test_get_email_valid(self):
        with patch('builtins.input', return_value="test@example.com"):
            self.assertEqual(get_email(), "test@example.com")

    def test_get_email_invalid(self):
        with patch('builtins.input', return_value="invalid_email"):
            with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
                get_email()
                self.assertEqual(mock_stdout.getvalue().strip(), "Invalid email address format.")

    def test_get_email_unexpected(self):
        with patch('builtins.input', return_value="test[at]example.com"):
            self.assertIsNone(get_email())

    def test_get_sql_query_valid(self):
        with patch('builtins.input', return_value="SELECT * FROM users"):
            self.assertEqual(get_sql_query(), "SELECT * FROM users")
    def test_get_sql_query_invalid(self):
        with patch('builtins.input', return_value="DROP TABLE users"):
            with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
                get_sql_query()
                self.assertEqual(mock_stdout.getvalue().strip(), "Invalid query. Destructive operations are not allowed.")

    def test_get_sql_query_unexpected(self):
        with patch('builtins.input', return_value="DELETE FROM users"):
            self.assertIsNone(get_sql_query())

if __name__ == '__main__':
    unittest.main()
```

This set of unit test cases covers valid, invalid, and unexpected scenarios for each of the input validation functions in the provided Python code. Running these tests using the unittest framework will help ensure the correctness and reliability of the input validation logic.