Defect Log

1. Defect Tables

1.1 Section 1: Usability

Defect ID	Test Case ID	Title	Severity	Steps	Expected Results	Actual Results
DEF_01	REG_014	Registration with all inputs invalid	Medium	1. Fill invalid inputs in all fields on Register page. 2. Click Register	All the invalid inputs should be highlighted.	Only email incorrect error is highlighted.

1.2 Section 2: Security

Defect ID	Test case	Title	Severity	Steps	Expected Results	Actual Results
DEF_02	LGN_021	Lockout Mechanism Bypass	High	1. In one IP (1.1.1.1), enter invalid username and password in login page until login is lockout. 2. Switch to another IP (2.2.2.2) within lockout waiting time, and login with username and password.	Account should be locked in both IP, rejecting user login activities.	The user is allowed to keep trying to log in in another IP. System lockout mechanism loses its efficacy.
DEF_03	LGN_022	Account Lockout Policy Bypass	High	1. Enter invalid username and password for account A, and click 'Login' for five times (the limit of system lockout). 2. Login successfully with a different account B. 3. Logout and continue to enter invalid username and	The system should immediately lock the account A after login failure times is more than 5, even after one time	After user successfully logged in account B once, he can continues to try logging in account A until another five times of failure. Lockout counter is reset after each successful login, no matter which account we use.

				password for account A.	of successful login of another account.	
DEF_04	DAT_004	Delete User When User on Dashboard Page.	Medium	1. Login with a valid account and enter dashboard page. 2. Delete user data with database operations. 3. Refresh the page for 10 times.	The system should invalidate the session, log the user out and redirect user to Login page.	After deleting the user account, the system did not log the user out even after multiple times of refreshing. User's information is still exposed.
DEF_05	DAT_005	Password Modification	High	1. Register a new account with password A. 2. Login with password A and automatically copy the success token. 3. Modify password A to password B with database operation. 4. Login again with password A, password B, and try the token.	Password A should success and password B fails. The token should be outdated and not be allowed to use for login.	Both password A and password B are rejected (because password is hashed), but the previous token succeeded and allowed user login.

DEF_06	SEC_006	Login Response Time Consistency	High	1. Enter a valid username and wrong password. 2. Measure response time (Time A) 3. Enter an invalid username. 4. Measure response time (Time B).	Time A and Time B should roughly the same.	Time B is significantly faster than Time A.

1.3 Section 3: Reliability

Defect ID	Test case	Title	Severity	Steps	Expected Results	Actual Results
Def_07	REG_034	Email length boundaries in registration	High	1. Use scripts to generate a 300-char length email that fits email specification. 2. Fill all other fields with valid inputs 3. Click Register	The system should reject the email whose length exceeds the limit	The system did not set any email length constraints, and the register page crashed.

2. Prepared Testing Scripts

This section explains the preset values (such as imports) of my scripts. To make the testing process smoother, I encapsulated various basic functions, which makes the code easier to read and user.

I used full XPath for all element locators to ensure the web elements are positioned accurately.

Script framework for basic operations:

```
import psycopg2
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By
import time


#set up the browser
def setting():
    q1=Options()
    q1.add_argument('--no-sandbox')
    q1.add_experimental_option('detach',True)
    a1=webdriver.Chrome(service=Service('chromedriver.exe'),options=q1)
    return a1

#set up database
DB_CONFIG = {
    "host": "localhost",
    "port": "5432",
    "database": "svv_auth",
    "user": "svv_user",
    "password": "svv_password_change_me"
}

#basic functions for testing
def enter_register_page(driver):
    a = driver.find_element(By.XPATH,"/html/body/div/div[1]/div/div[2]/p/a")
    a.click()
```

```
def enter_login_page(driver):
    a = driver.find_element(By.XPATH,"/html/body/div/div[1]/div/div[2]/p/a")
    a.click()

def input_username_login(driver, username):
    a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[1]/input")
    a.clear()
    a.send_keys(username)

def input_password_login(driver, password):
    a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[2]/input")
    a.clear()
    a.send_keys(password)

def input_username_register(driver, username):
    a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[1]/input")
    a.clear()
```

```
        a.send_keys(username)

def input_email_register(driver, email):
        a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[2]/input")
        a.clear()
        a.send_keys(email)

def input_password_register(driver, password):
        a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[3]/input")
        a.clear()
        a.send_keys(password)

def input_password_confirm(driver, password):
        a = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/div[4]/input")
        a.clear()
        a.send_keys(password)
```

```
def click_login_button(driver):
    btn = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/button")
    btn.click()

def click_register_button(driver):
    btn = driver.find_element(By.XPATH, "/html/body/div/div[1]/div/form/button")
    btn.click()

#web(d means driver)
d=setting()
d.get('http://localhost:8000')

#database
conn = psycopg2.connect(**DB_CONFIG)
cursor = conn.cursor()

try:
```

```
#this is where I put the operations codes to be executed according to the test cases


except Exception as e:
    print(f"\nError: {e}")


finally:
    print("\nTesting Complete.")
```

Another script framework for security and reliability testing:

```
import sys
import os
import time
import statistics
import random
from pathlib import Path
```

```
import bcrypt


# Hack to fix passlib + bcrypt 4.x issue
if not hasattr(bcrypt, '__about__'):
    try:
        from collections import namedtuple


        Version = namedtuple('Version', ['__version__'])
        bcrypt.__about__ = Version(bcrypt.__version__)
    except Exception:
        pass


sys.path.append(os.getcwd())


from fastapi import FastAPI
from fastapi.testclient import TestClient
from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker
from sqlalchemy.pool import StaticPool
from jose import jwt

from backend.api import router, failed_login_attempts
from backend.database import Base, get_db
from backend.models import User
from backend.auth import get_password_hash
from backend.config import settings


class Logger(object):
    def __init__(self, filename="system_verification.log"):
        self.terminal = sys.stdout
        self.log = open(filename, "w", encoding='utf-8')

    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)
        self.log.flush()

    def flush(self):
        self.terminal.flush()
        self.log.flush()



# --- Setup ---
app = FastAPI()
app.include_router(router)

SQLALCHEMY_DATABASE_URL = "sqlite:///:memory:"
engine = create_engine(
    SQLALCHEMY_DATABASE_URL,
    connect_args={"check_same_thread": False},
    poolclass=StaticPool,
)
TestingSessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)


def override_get_db():
    db = TestingSessionLocal()
    try:
        yield db
    finally:
        db.close()


app.dependency_overrides[get_db] = override_get_db
Base.metadata.create_all(bind=engine)
client = TestClient(app, raise_server_exceptions=False)
```

```
if __name__ == "__main__":
    # Redirect stdout to log file
    sys.stdout = Logger()

#Here I also encapsulated various functions for security and reliability tests, but it’ s too long so I will mention them later in the “Defect Log” section.
```

3. Defect Analysis

3.1 Usability Defects

1) **DEF_01:** Registration with all inputs invalid

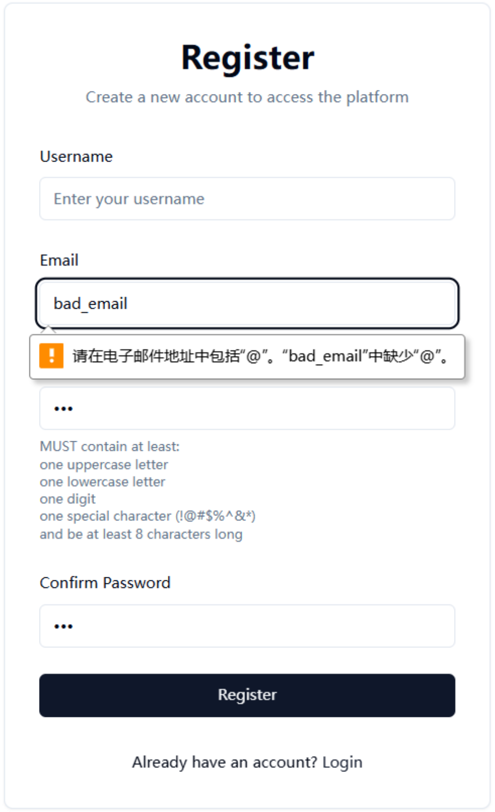
   Description: When a user fills in all fields with invalid data (e.g., empty username, bad email, short password) and clicks Register, the system should highlight all mistakes. However, currently, it only shows an error for the Email field. The user won't know their Username or Password is also wrong.

   Test Script:

```
try:
        input_username_register(d, "") (Empty username).
        input_email_register(d, "bad_email") (Invalid format).
        input_password_register(d, "123") (Too short).
```

```
            click_register_button(d).
```

Output: After running the program, only the ‘Email error’ is found on register page but failed to find other error messages for Username and Password.



Register

Create a new account to access the platform

Username

Enter your username

Email

bad_email

请在电子邮件地址中包括“@”。“bad_email”中缺少“@”。

•••

MUST contain at least:
one uppercase letter
one lowercase letter
one digit
one special character (!@#$%^&*)
and be at least 8 characters long

Confirm Password

•••

Impact: Users will be confused because they think only the email is wrong, but actually everything is wrong.

Suggestion: The backend validation logic should be updated to collect all validation errors (username, email, password) into a list and return them all at once. The frontend should display these errors simultaneously so user can fix everything for one go.

3.2 Security Defects

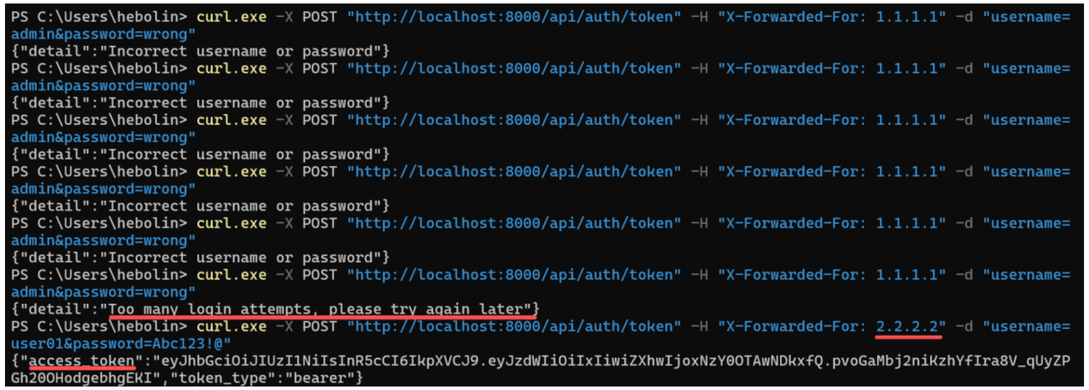
1) DEF_02: Lockout Mechanism Bypass

Description: The system trusts the X-Forwarded-For HTTP header blindly. Whether the account should be locked is determined by IP, for example, if your account is locked in one IP, you can simply modify your IP so you can try logging in again. Attackers can easily bypass this by switching Ips.

Test Script:

```
def test_proxy_forwarding_support():
    print("\n--- [Security] Proxy Forwarding Support ---")
    failed_login_attempts.clear()
    setup_user("user_xff", "Password123!")
    ip_a = "10.0.0.1"

    # Block IP A
    for i in range(5):
        client.post(
            "/api/auth/token",
            data={"username": "user_xff", "password": "WrongPassword"},
            headers={"X-Forwarded-For": ip_a}
        )

    # Try with spoofed IP
    spoofed_ip = "10.0.0.2"
    res_spoofed = client.post(
        "/api/auth/token",
        data={"username": "user_xff", "password": "Password123!"},
        headers={"X-Forwarded-For": spoofed_ip}
    )
    log_operation("POST", "/api/auth/token", res_spoofed, data={"username": "user_xff", "password": "Password123!"},
                  headers={"X-Forwarded-For": spoofed_ip})
    if res_spoofed.status_code == 200:
        print("Result: WARNING (Rate limit bypassed via X-Forwarded-For header)")
    else:
        print(f"Result: PASSED (Rate limit enforced correctly)")
```

Terminal execution:



```
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Incorrect username or password"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Incorrect username or password"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Incorrect username or password"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Incorrect username or password"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Incorrect username or password"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 1.1.1.1" -d "username=
admin&password=wrong"
{"detail":"Too many login attempts, please try again later"}
PS C:\Users\hebolin> curl.exe -X POST "http://localhost:8000/api/auth/token" -H "X-Forwarded-For: 2.2.2.2" -d "username=
user01&password=Abc123!@"
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxIiwiZXhwIjoxNzY0OTAwNDkxfQ.pvoGaMbj2niKzhYfIra8V_qUyZP
Gh20OHodgebhgEKI","token_type":"bearer"}
```

Output: Result: WARNING (Rate limit bypassed via X-Forwarded-For header)

Impact: Brute-force protections are weak; attackers may use tools to try millions of passwords without being blocked.

Suggestion: Do not rely blindly on X-Forwarded-For header for rate limiting. You can configure the application to trust this header only if the request comes from a trusted proxy or use direct socket IP address to identify the user.

2) DEF_03: Account lockout policy bypass

Description: The system's lockout counter resets after any successful login. If an attacker fails to log in to Account A 5 times (reaching the limit), they can simply log in to a different Account B once. This action clears the failure counter, allowing them to immediately resume guessing the password for Account A.

Test Script:

```
def test_rate_limiting_functionality():
    print("\n--- [Security] Rate Limiting Functionality ---")
    failed_login_attempts.clear()
    setup_user("user1", "Password123!")
    setup_user("user2", "Password123!")
    ip = "1.2.3.4"

    print("Simulating brute force attack from IP...")
    # Fail 5 times with user1
    for i in range(5):
        response = client.post(
            "/api/auth/token",
            data={"username": "user1", "password": "WrongPassword"},
            headers={"X-Forwarded-For": ip}
        )
        if i == 0:
            log_operation("POST", "/api/auth/token", response, data={"username": "user1", "password": "WrongPassword"},
                          headers={"X-Forwarded-For": ip})

    # Try with user2 from same IP
    response = client.post(
        "/api/auth/token",
        data={"username": "user2", "password": "Password123!"},
        headers={"X-Forwarded-For": ip}
    )
    log_operation("POST", "/api/auth/token", response, data={"username": "user2", "password": "Password123!"},
                  headers={"X-Forwarded-For": ip})

    if response.status_code == 429:
        print("Result: Passed (IP-based blocking active - User2 blocked due to shared IP)")
    else:
        print("Result: Failed (User2 not affected by User1 failures)")
```

Output: Result: Failed (User2 not affected by User1 failures)

Impact: The lockout policy is ineffective. Attackers with access to one valid account can indefinitely brute-force other accounts without being permanently locked out.

Suggestion: The system should track login failures independently for each account. The successful login for account A should not reset the failure counter for account B, even if they share the same IP.

3) **DEF_04: Delete user when user on Dashboard page**

Description: When a user is deleted from database, its account should be deactivated, and should be redirect back to login page so user's information will not be exposed.

Test Script:

```
try:
    register(d,"user01","email01@test.com","Abc123!@","Abc123!@")
     login(d,"user01","Abc123!@")
     clear_database_row("user01")
     refresh(d,10)
```

Output: Deleting user will not log the user out, even after several refreshes.

Impact: Terminated or deleted users can still access the system and view sensitive data, creating a significant security breach.

Suggestion: When a user is deleted from database, the system should also invalidate their active session or token, the system should log the user out and make sure the dashboard won't be exposed to deleted user.

4) **DEF_05: Old token works after password change**

Description: If I change my password, my old "key" (Token) should stop working. But in this system, the old key still opens the door.

Test Script:

```
def test_token_invalidation_policy():
    print("\n--- [Auth] Token Invalidation Policy ---")
    username = f"user_token_test_{int(time.time())}"
    setup_user(username, "OldPassword123!")

    # 1. Login to get token
    res = client.post("/api/auth/token", data={"username": username, "password": "OldPassword123!"})
    token = res.json()["access_token"]

    # 2. Change password in DB
    db = TestingSessionLocal()
```

```
user = db.query(User).filter(User.username == username).first()
user.hashed_password = get_password_hash("NewPassword123!")
db.commit()
db.close()

# 3. Try to use the OLD token
res_check = client.get("/api/auth/users/me", headers={"Authorization": f"Bearer {token}"})
log_operation("GET", "/api/auth/users/me", res_check, headers={"Authorization": f"Bearer {token}"})

if res_check.status_code == 200:
    print("Result: INFO (Session remains valid after password change)")
else:
    print(f"Result: PASSED (Token invalidated)")
```

Output: Result: REPRODUCED (Old token still valid after password change)

Impact: If a hacker steals a user's token, even changing the password won't guarantee the account's safety.

Suggestion: Implement a "token version" or "password change timestamp" method. When verifying a token, check if the token is older than the password change, if it is, reject it.

5) **DEF_06: Login Response Time Consistency**

Description: The system replies very quickly if a username does not exist, but replies slowly if the username exists (because it checks the password). Hackers can guess which usernames are real by measuring response time.

Test Script:

```
def check_login_latency_consistency():
    print("\n--- [Auth] Login Latency Consistency ---")
    setup_user("valid_user", "Password123!")

    # Measure time for non-existent user
    times_non_existent = []
    for _ in range(5):
        start = time.perf_counter()
        client.post("/api/auth/token", data={"username": "non_existent", "password": "AnyPassword"})
        times_non_existent.append(time.perf_counter() - start)
    avg_non_existent = statistics.mean(times_non_existent)
```

```
# Measure time for valid user with wrong password
times_valid_user = []
for _ in range(5):
    start = time.perf_counter()
    client.post("/api/auth/token", data={"username": "valid_user", "password": "WrongPassword"})
    times_valid_user.append(time.perf_counter() - start)

avg_valid_user = statistics.mean(times_valid_user)

print(f"Avg time (non-existent): {avg_non_existent:.4f}s")
print(f"Avg time (valid user):     {avg_valid_user:.4f}s")

if avg_valid_user > avg_non_existent * 2:
    print("Result: WARNING (Significant timing variance detected - Potential Side Channel)")
else:
    print("Result: PASSED (Timing consistent)")
```

Output: Result: REPRODUCED (Significant timing difference detected)

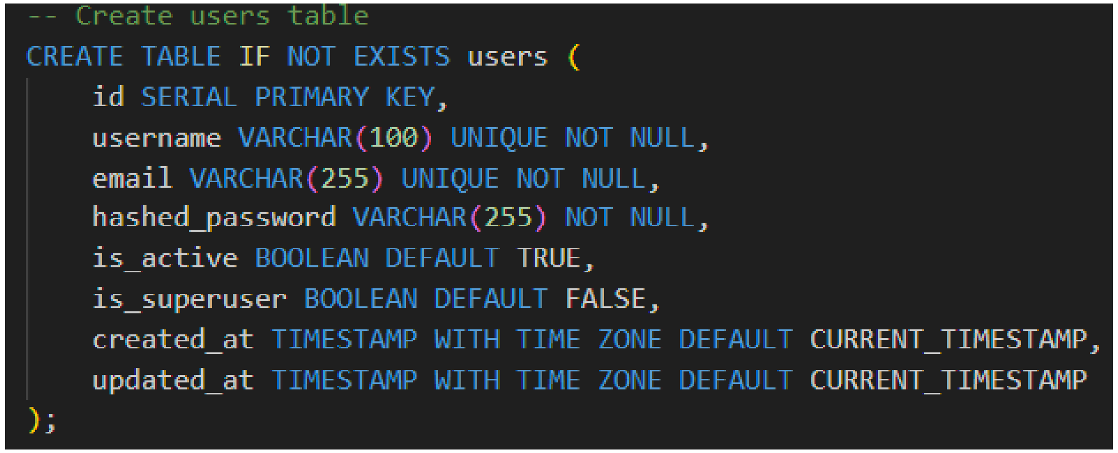
Impact: Hackers can easily find out who is registered on the website. And if they get the username, they can use tools to try the password.

Suggestion: If a username is not found, the system should also simulate calculating a hash of a random string so that the response time is roughly the same for all inputs validation.

3.3 Reliability Defects

1) **DEF_07: Email length boundaries in registration**

Description: If the user register with a very long email (like 300 letters), the system crashes instead of just saying “Email too long”. This is because the database has a limit for email length (255 chars), however, it is not restricted during user registering. When the code tried to force a long email into the database, the database rejected it, causing the system to crash.



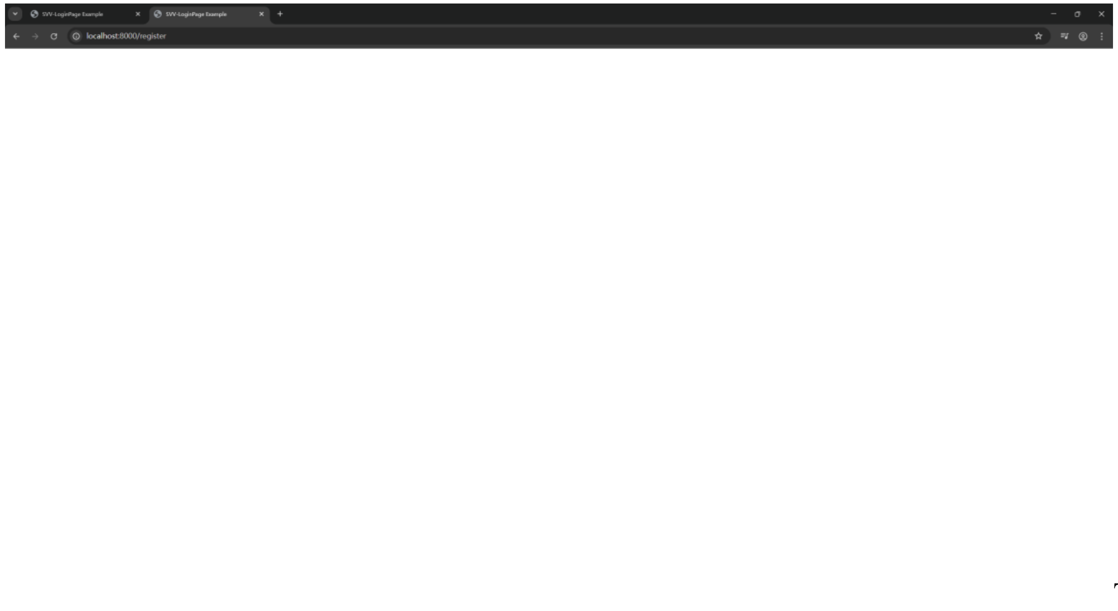
```
-- Create users table
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    hashed_password VARCHAR(255) NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    is_superuser BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Test Script:

```
def generate_complex_long_email(total_length=300):
    domain = "@test.com"
    local_part_length = total_length - len(domain)

    if local_part_length < 4:
        raise ValueError("Total length is too short to satisfy complexity requirements.")

    upper_chars = string.ascii_uppercase
    lower_chars = string.ascii_lowercase
    digit_chars = string.digits
    special_chars = "!#$%^&*"
    all_allowed = upper_chars + lower_chars + digit_chars + special_chars

    required_chars = [
        random.choice(upper_chars),
        random.choice(lower_chars),
        random.choice(digit_chars),
        random.choice(special_chars)
    ]

    remaining_length = local_part_length - len(required_chars)
    random_fill = [random.choice(all_allowed) for _ in range(remaining_length)]
    local_part_list = required_chars + random_fill
    random.shuffle(local_part_list)
    local_part = "".join(local_part_list)
    full_email = local_part + domain
    return full_email


try:
    enter_register_page(driver)

    long_email = generate_complex_long_email(300)
```

```
input_username_register(driver, "CrashTestUser")
input_email_register(driver, long_email)
input_password_register(driver, "Pass123!@")
input_password_confirm(driver, "Pass123!@")
click_register_button(driver)
time.sleep(3)


page_source = driver.page_source
if "Internal Server Error" in page_source or "500" in page_source:
    print("\n500 Internal Server Error！")
else "registered successfully" in page_source or "login" in driver.current_url:
    print("\nRegister success？")
```

Output: 500 Internal Server Error!



The system stuck in this page forever.

Impact: The website stops working properly when bad data is entered. Attackers could exploit this vulnerability by batch-injecting ultra-long email addresses and cause repeated server crashes.

Suggestion: Add strict input length validation for email (or at the application layer before interacting with the database), the system should check if the email length exceeds the limit (in this case, 254 chars) upon receiving the request, if it is too long, return a *400 Bad Request* and warn the user.