from flask import Flask, request

from jinja2 import Template

import os as os

app = Flask(\_\_name\_\_)

@app.route("/page")

def page():

    name = request.values.get('name')

    age = request.values.get('age', 'unknown')

output = Template('Hello ' + name + '! Your age is ' + age + '.').render()

    return output

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

Фрагмент кода

11 строка «output = Template('Hello ' + name + '! Your age is ' + age + '.').render()» - является уязвимой к инъекциям со стороны юзера, что открывает доступ к выполнению произвольного кода на сервере для манипулирования логикой.  
Это открывает доступ к множеству уязвимостей таких как:

* SQL Injections
* XSS
* SSTI

В нашем примере наглядно будет продемонстрировать XSS и SSTI. Так как значения name и age произвольны, и юзер может манипулировать ими напрямую без фильтрации ввода он может напрямую взаимодействовать с сервером и выполнить любой желаемый ему код.  
Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Пример эксплуатации SSTI

Как мы видим, операция умножения выполнилась, что говорит о возможности взаимодействовать напрямую с логикой сервера изнутри запроса: output = Template('Hello ' + name + '! Your age is ' + age + '.').render()

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание  
Пример другой простенькой операции SSTI

Отвечая на вопрос «к каким последствиям может привести эксплуатация **SSTI** уязвимости?» хочу подметить:

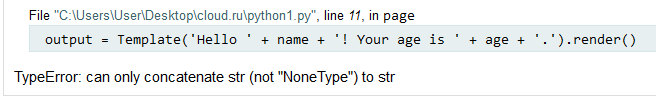
* Это наиболее критический риск, так как **атакующий может получить контроль над сервером**, на котором выполняется приложение. ({{ ‘rm -rf /’ }} и т.д.)
* Атакующий может получить доступ к базе данных и другим чувствительным ресурсам сервера.
* Можно запустить процессы, которые **угрожают доступности приложения**, например, путем **истощения ресурсов сервера. (DoS)**

Также, необходимо подметить уязвимость межсайтового скриптинга (XSS), из-за отсутствия надлежащей валидации ввода, мы можем написать обычный JavaScript отрывок, который будет выполняться по пожеланию юзера.  
Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание  
Пример эксплуатации XSS

Отвечая на вопрос «к каким последствиям может привести эксплуатация XSS уязвимости?» хочу подметить:

* Атакующий **может украсть сессионные куки** и использовать их для установления сессии как жертва, **обходя процессы аутентификации**.
* Возможность **изменения отображаемого содержимого на стороне клиента** для **введения пользователя в заблуждение или вредоносных действий**.
* **Кража пользовательских данных, доступ к нужному управлению сайтом.**

Справедливо утверждать, что при наличии подключенной базы данных и идентичной логике запросов злоумышленник мог бы написать свой SQL query для взаимодействия с базой данных.  
  
Также, из-за 15 строки, “app.run(debug=True)”, которая говорит о включенном режиме отладки в потенциально «производственной» среде, может произойти утечка информации об инфраструктуре приложения атакующим.  
 Отрывок из отладчика при ошибке

  
Подобные ошибки могут дать злоумышленнику понять логику запроса значительно упростив задачу и раскрыть информацию об устройстве всей инфраструктуры.  
  
Также, стоит подметить, возможность потенциальных IDOR и SQL Injection уязвимостей при подобном подходе написании кода, так как у нас нет проверке ввода или наличия сессий юзера, что позволяет получить доступ к потенциальному «аккаунту» другого юзера подставив в поле «name» и «age» нужные нам параметры (IDOR, строка 8). Что касается SQL инъекции (строка 11) – та же самая логика что и с SSTI и XSS, которая может привести к утечке чувствительной информации и неправомерном доступе к базам данным и их изменению и т.д.  
  
Для исправления вышеуказанных уязвимостей по порядку:

* XSS, SSTI , possible SQL Injection – строка 11 должна быть изменена по типу:   
  return render\_template\_string("Hello {{ name }}! Your age is {{ age }}.", name=user\_data["name"], age=user\_data["age"])  
  Мы используем Flask render\_template или render\_template\_string с безопасной передачей параметров экранируя переменные name и age делая их безопасными для отображения предотвращая возможность XSS атак и минимизируя шанс на SSTI. Тем не менее, хочется подметить такие способы как: избегание динамической генерации шаблонов, санитизацию ввода используя “Bleach”.
* Изменить строчку app.run(debug=True) на app.run(debug=False), что предотвратит вывод отладочной информации в продакшне.
* Создать логику сессий и секретных ключей для того чтобы избежать IDOR уязвимости, использование и контроль id для юзеров для каждой сессии.

Не стал учитывать фильтрации на условные кавычки как способ решения и тому подобные способы в этом контексте, так как они легко обходятся и, по моему мнению, лучше подходить к этому со стороны общей безопасности не наслаивая «костыли», которые не гарантируют безопасности.