GS CyberSecurity

Vulnerabilidades amostras, e como consertar e detectar em pipelines CI/CD

Nome: Bruno Eduardo Caputo Paulino

Rm:558303

Turma: 3ESPW

Fiap 2025

1. Introdução

Este documento apresenta quatro vulnerabilidades amplamente reconhecidas. O objetivo é demonstrar como cada falha funciona, como pode ser explorada, como corrigi-la e de que forma um pipeline de CI/CD moderno — utilizando SAST, DAST e SCA — pode identificá-la antes que a aplicação seja implantada em produção.

2. SQL Injection

SQL Injection ocorre quando a aplicação insere diretamente dados do usuário dentro de um comando SQL. Isso permite que o atacante modifique a query original, resultando em vazamento de dados, exclusão de registros ou comprometimento completo do banco. É uma das vulnerabilidades mais críticas devido ao alto impacto.

2.1 Código Vulnerável

```
@app.route("/sql-injection-vulnerable")
def sql_injection():
    name = request.args.get("name", "")
    query = f"SELECT * FROM users WHERE name = '{name}'"
    conn = sqlite3.connect("test.db")
    cur = conn.cursor()
    cur.execute(query)
    return "infectado"
```

2.2 Como o Ataque Funciona

O atacante utiliza payloads que manipulam a lógica SQL, como 'OR 1=1', forçando retornos indevidos.

```
import requests

payload = "admin' OR '1'='1"
r = requests.get("http://localhost:5000/sql-injection-vulnerable?name="
+ payload)
print(r.text)
```

2.3 Código Corrigido

A correção exige o uso de prepared statements (queries parametrizadas).

```
@app.route("/sql-injection-safe")
def sql_injection_safe():
    name = request.args.get("name", "")
    query = "SELECT * FROM users WHERE name = ?"
    cur.execute(query, (name,))
    return "ok"
```

2.4 Detecção em CI/CD

• SAST: Detecta concatenação insegura de SQL.

• DAST: Executa ataques automáticos com payloads SQL Injection.

• SCA: Verifica CVEs em bibliotecas de banco.

3. Quebra de Controle de Acesso

A falha ocorre quando a aplicação confia que o usuário informará seu próprio nível de permissão. Isso permite que atacantes simplesmente modifiquem parâmetros e assumam privilégios elevados.

3.1 Código Vulnerável

```
@app.route("/acesso-vulneravel")
def acesso_vulneravel():
    role = request.args.get("role")
    if role == "admin":
        return "infectado"
    return "ok"
```

3.2 Ataque

/acesso-vulneravel?role=admin

3.3 Código Corrigido

```
@app.route("/acesso-seguro")
def acesso_seguro():
    token = request.headers.get("X-Token")
    if token != "admin-token":
        return "ok"
    return "ok"
```

3.4 Detecção em CI/CD

• SAST: Detecta endpoints sem autenticação.

• DAST: Testa manipulação de parâmetros de permissão.

• SCA: Apenas verifica bibliotecas vulneráveis nada expecifico para esta vulnerabilidade.

4. Desserialização Insegura

Desserialização insegura ocorre quando dados binários são reconstruídos em objetos sem validação adequada. O módulo Python 'pickle' é perigoso pois permite execução arbitrária de código.

4.1 Código Vulnerável

```
@app.route("/deserializacao-vulneravel", methods=["POST"])
def deserializacao_vulneravel():
    payload = request.data
    obj = pickle.loads(payload)
    return "infectado"
```

4.2 Ataque

```
class Exploit:
    def __reduce__(self):
        return (os.system, ("calc",))
```

4.3 Código Corrigido

```
@app.route("/deserializacao-segura", methods=["POST"])
def deserializacao_segura():
    data = json.loads(request.data)
    return "ok"
```

4.4 Detecção em CI/CD

• SAST: Sinaliza uso de pickle como crítico.

• DAST Executa payloads de desserialização.

• SCA: Apenas verifica bibliotecas vulneráveis nada expecifico para esta vulnerabilidade..

5. Injeção de Comando

Injeção de comando ocorre quando um comando de sistema operacional é executado usando entrada do usuário. Com 'shell=True', o atacante consegue executar comandos arbitrários.

5.1 Código Vulnerável

```
@app.route("/comando-vulneravel")
def comando_vulneravel():
    cmd = request.args.get("cmd", "")
    output = subprocess.check_output(cmd, shell=True)
    return "infectado"
```

5.2 Código Corrigido

```
@app.route("/comando-seguro")
def comando_segur():
    allowed = ["date", "whoami"]
    cmd = request.args.get("cmd", "")
    if cmd not in allowed:
        return "ok"
    output = subprocess.check_output(["/usr/bin/" + cmd])
    return "ok"
```

5.3 Detecção em CI/CD

• SAST: Detecta uso de 'shell=True'.

• DAST: Testa comandos encadeados (';', '|', '&&').

• SCA: Apenas verifica bibliotecas vulneráveis nada expecifico para esta vulnerabilidade.

6. Conclusão

As vulnerabilidades analisadas representam riscos reais e sérios. O uso combinado de SAST, SCA e DAST dentro de um pipeline CI/CD moderno assegura que falhas sejam detectadas antes de chegarem ao ambiente produtivo.