Manual do Projeto

Tatiane de Matos Silva

Henrique Christopher de Castro Leão

Manual do projeto: Testes de Vulnerabilidades Web

Este documento apresenta a visão geral da arquitetura de rede implementada no ambiente do cliente, descrevendo cada componente, suas funções e como eles se relacionam.

A plataforma permite a visualização, acompanhamento e compreensão de como funcionam ataques controlados em um ambiente seguro. Lembrando que, todas as máquinas alvo estão isoladas e não afetam sua infraestrutura real.

1. Componentes da arquitetura

1. PfSense – Firewall e Roteador

- Recebe a Internet pela interface WAN (em0)
- Gerencia a rede LAN (em1)
- Gerencia a rede OPT1 (em2) – Host-Only

Atua como:

- Firewall
- Roteador
- Servidor DHCP para a rede 192.168.200.0/24
- Controlador de isolamento das redes internas

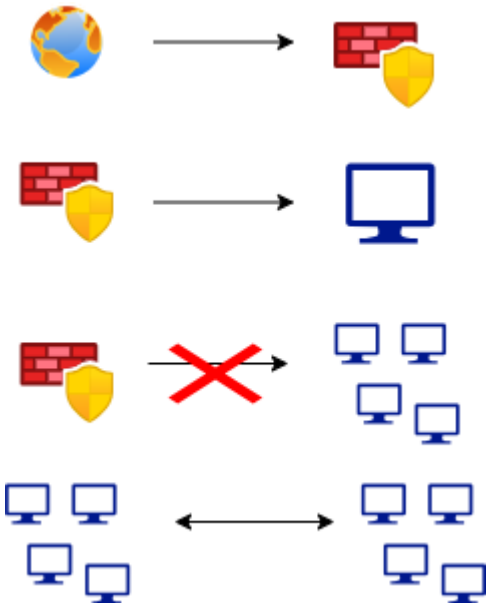
2. Host Físico (computador do cliente) – Rede Host-Only

- Conectado à interface OPT1 do pfSense.
- Recebe IP automaticamente (ex.: 192.168.200.100).
- Acesso administrativo ao pfSense.

3. Máquinas Virtuais – Rede Interna "labvirtual"

- Ambiente totalmente isolado.
- Comunicação apenas entre máquinas virtuais.
- Não possuem rota de saída para o pfSense nem para o Host.
- Usadas para testes e simulação de ataques controlados.

Fluxo de Rede Resumido



Internet → pfSense (WAN)
pfSense (OPT1) → Host físico

pfSense NÃO se comunica com as VMs e as VMs só há comunicação entre si, pois estão em rede isolada

2. Entendendo a Rede Host-Only

A rede Host-Only, é uma rede isolada; pfSense e Host Físico (não tem internet direta).

Serve para:

- Administração do pfSense
- Captura de pacotes
- Ferramentas de diagnóstico

Faixa de rede

- 192.168.200.0/24
- Gateway: 192.168.200.1 (pfSense)
- DHCP: 192.168.200.10 – 192.168.200.200

Benefícios:

- Segurança total: não expõe o pfSense na LAN real
- Canal exclusivo de administração
- Evita que as VMs tenham acesso acidental ao firewall

3. Tutorial – Como Acessar o pfSense

Os pré-requisitos é estar conectado na rede Hosty-Only e ter um navegador instalado

1. Verificar o IP do Host:

No Windows, digitar no CMD o comando “ipconfig” vai aparecer “Adaptador Host-Only IPv4: 192.168.200.100

2. Acessar o pfSense:

Abra o navegador e digite: https://192.168.200.1. O primeiro login é padrão: Admin e pfsense.

O cliente pode visualizar:

- Logs de ataque
- Regras de firewall
- Relatórios de conexão
- Estatísticas de tráfego

3. Dentro do pfSense

Você verá a tela de dashboard, como: Status do sistema, tráfego, interfaces, regras de firewall, monitoramento, logs de conexões, entre outros.

4. Ambiente das máquinas virtuais

1. O cliente poderá visualizar:

- Máquina atacante
- Máquinas vulneráveis
- Processos e logs
- Resultados dos testes de segurança

Tudo isso dentro do VirtualBox.

2. Dentro do VirtualBox

O cliente verá uma lista como:

- Firewall – pfSense

- AttackBox (máquina que fara o ataque)
- VM1 – Servidor Vulnerável
- VM2 – Aplicação Vulnerável
- VM3 – Serviço Auxiliar Vulnerável

3. Acessar as máquinas

Cada máquina tem um IP fixo na rede isolada:

Máquina	**IP**	**Função**
AttackBox	10.0.0.200	Executa os ataques
VM1	10.0.0.101	Servidor vulnerável
VM2	10.0.0.102	Servidor vulnerável
VM3	10.0.0.103	Serviço vulnerável

O cliente pode executar ataques controlados (com o script), ver logs, monitorar respostas, repetir testes, fazer demonstrações internas.

5. Tutorial - Como será a execução dos testes

Passo 1 — Ligar pfSense

Isso ativa a rede host-only e distribui os IPs internos.

Passo 2 — Ligar Máquina Atacante (AttackBox)

Ela já contém o script configurado.

Passo 3 — Ligar as Máquinas Vulneráveis

Elas respondem às conexões da AttackBox.

Passo 4 — Rodar o Script de Testes

Na AttackBox, o cliente executa: ./scan_vulnerabilidades.sh

O cliente verá: Portas abertas, serviços ativos, respostas das VMs, identificação de falhas simuladas

Passo 5 — Ver resultados no Firewall

No painel do pfSense, o cliente pode ver: Conexões de entrada, logs de ataques simulados, regras que permitiram ou bloquearam tráfego, gráficos e estatísticas

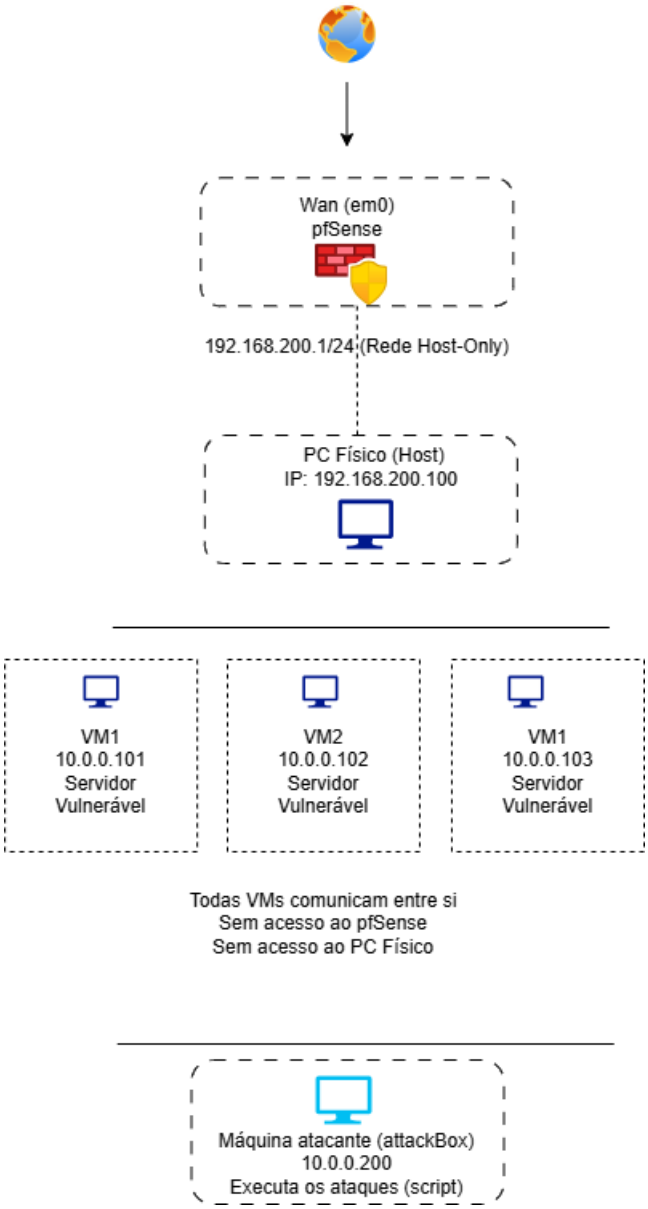
6. Segurança e entrega

O ambiente é totalmente isolado:

- Sem acesso à internet pelas VMs
- Sem risco de vazamento
- Sem impacto na rede real do cliente
- Os ataques ocorrem somente na rede LABVIRTUAL

O cliente receberá:

- O computador/servidor já configurado
- Todas as VMs funcionando
- Script de ataques pré-instalado
- Manual em PDF
- Acesso ao firewall
- Suporte para dúvidas e manutenção



Funções e Relações Entre os Componentes

pfSense (Firewall)

Atua como divisor entre a rede do cliente e a rede de laboratório, permite monitorar ataques gerados pelas VMs. Garantia de que nada sai da rede interna para a internet e só o PC físico pode acessar o pfSense.

PC Físico (Host)

É a estação de controle do cliente, que conecta no pfSense para: ver logs, monitorar os ataques, acompanhar o comportamento da rede.

Rede Interna “labvirtual”

Isolada totalmente do mundo externo. As VMs se comunicam entre si e nunca acessam o firewall e éonde ocorrem os testes e ataques.

VMs Vulneráveis

Simulam serviços com falhas reais e são os alvos do script de ataque.

AttackBox

Máquina responsável por executar as varreduras, testes de intrusão, scripts de exploração, coleta de evidências.

E como isso acontece? Esta seção deve ser enviada ao cliente final. Tudo explicado sem termos técnicos complexos, mas de fácil entendimento.

7. Segurança e recomendações

- Salvar backups do pfSense sempre antes de mudanças significativas.
- Coloque descrições nas regras do firewall.
- Conhecimento das principais vulnerabilidades web

8. Requisitos Técnicos

Requisitos de Software:

- Python ou C#
- VitualBox
- PfSense
- Suricata

Requisitos de Hardware:

- Processador compatível com virtualização
- Espaço livre em disco para Máquinas Virtuais (+/- 30 GB)

9. Sobre o Código

O script **`zap_passive_report.py`** é uma ferramenta segura de análise passiva de vulnerabilidades, projetada para ser usada **exclusivamente em ambiente de laboratório**. Ele utiliza o *OWASP ZAP* para realizar **coleta de informações e análise passiva**, sem executar ataques ativos.

- Conecta ao OWASP ZAP
- Faz um *Spider* para descobrir páginas do site alvo
- Aguarda a finalização do *Passive Scan*
- Coleta todos os alertas encontrados
- Gera relatórios em **JSON** e **CSV**
- Classifica alertas por nível de risco

1. Conexão com o OWASP ZAP

A função `connect_zap()` abre uma conexão entre o script e o OWASP ZAP usando proxy na porta 8080.

Isso permite que o Python controle o ZAP:

- enviar comandos
- puxar alertas
- iniciar varreduras

Se o ZAP não estiver aberto, o script encerra automaticamente.

2. Spider Scan (Mapeamento de Páginas)

A função `spider_scan()`:

- manda o ZAP visitar o site alvo (`TARGET`)
- coleta links internos
- mapeia o site sem atacar

É uma navegação automatizada que descobre páginas e parâmetros.

O script acompanha o progresso até 100% ou até atingir um *timeout* de segurança.

3. Passive Scan (Análise Passiva)

A função `passive_scan()`:

- aguarda o ZAP analisar o tráfego capturado pelo Spider
- essa análise não envia payloads maliciosos
- verifica cabeçalhos, cookies, boas práticas de segurança, etc.

Só técnicas **não invasivas** são usadas.

O script aguarda até um limite configurado (*PASSIVE_TIMEOUT*) para evitar travamentos.

4. Coleta de Alertas

A função `get_alerts()`:

Solicita todos os alertas gerados pelo ZAP, filtra apenas os da URL alvo, devolve uma lista com detalhes como:

→ nome da vulnerabilidade
→ nível de risco (Low/Medium/High)
→ URL afetada
→ parâmetros envolvidos
→ evidências encontradas

5. Exportação de Relatórios

O script gera automaticamente:

- JSON (zap_passive_alerts.json)
  - ❖ completo
  - ❖ estruturado
  - ❖ útil para auditoria

- CSV (zap_passive_alerts.csv)
  - ❖ compatível com Excel
  - ❖ fácil para gerar gráficos ou planilhas

Este script é considerado SEGURO

- Não faz Active Scan
- Não dispara payloads
- Não explora falhas
- Apenas coleta e analisa tráfego normal
- Totalmente indicado para ambientes sensíveis

6. Como integrar esse script ao seu projeto

Rodar o ZAP em modo daemon no host do laboratório:

- Abra ZAP ou execute zap.sh -daemon -host 127.0.0.1 -port 8080 (usar documentação ZAP).
- Ajustar TARGET para seu ambiente de laboratório (juice-shop.lab ou IP interno).
- Executar o script (indicando ambiente isolado):
- python3 zap_passive_report.py

Os arquivos zap_passive_alerts.json e zap_passive_alerts.csv serão gerados, incorpore esses relatórios no processo de documentação e validação.

10. Código

""" zap_full_scan_robusto.py

⚠️ Execute APENAS em ambiente de laboratório com autorização explícita.

Este script: • Verifica se o ZAP está online • Faz Spider • Roda Passive Scan • Roda Active Scan (Injection, XSS, SSRF, Broken Access Control) • Gera relatórios detalhados em JSON e CSV • Mostra estatísticas resumidas por risco e categoria OWASP

"""

from zapv2 import ZAPv2 from time import sleep from pprint import pprint import json import csv import sys

===== CONFIGURAÇÃO =====

ZAP_ADDRESS = '127.0.0.1' ZAP_PORT = '8080' TARGET = 'http://juice-shop.lab/' # URL do seu laboratório API_KEY = None # ou sua API key, se estiver configurada

===== FUNÇÃO: Conectar ao ZAP =====

def connect_zap(): try: zap = ZAPv2(apikey=API_KEY, proxies={'http': f'http://{ZAP_ADDRESS}:{ZAP_PORT}', 'https': f'http://{ZAP_ADDRESS}:{ZAP_PORT}'}) # Teste de conexão version = zap.core.version print(f"[+] Conectado ao ZAP (versão {version})") return zap except Exception as e: print(f"[ERRO] Não foi possível conectar ao ZAP: {e}") sys.exit(1)

===== FUNÇÃO: Spider =====

def spider_scan(zap, target): print(f"\n[+] Iniciando Spider em {target}") scan_id = zap.spider.scan(target) while int(zap.spider.status(scan_id)) < 100: print(f"Spider: {zap.spider.status(scan_id)}%") sleep(2) print("[+] Spider finalizado")

===== FUNÇÃO: Passive Scan =====

def passive_scan(zap): print("\n[+] Iniciando Passive Scan") while int(zap.pscan.records_to_scan) > 0: print(f"Passive scan restante: {zap.pscan.records_to_scan}") sleep(2) print("[+] Passive Scan finalizado")

===== FUNÇÃO: Active Scan =====

def active_scan(zap, target): print("\n[+] Iniciando Active Scan (Injection, XSS, SSRF, BAC...)") scan_id = zap.ascan.scan(target) while int(zap.ascan.status(scan_id)) < 100: print(f"Active scan: {zap.ascan.status(scan_id)}%") sleep(5) print("[+] Active Scan concluído")

===== FUNÇÃO: Coletar alertas =====

def get_alerts(zap, target): alerts = zap.core.alerts(baseurl=target) print(f"[+] Total de alertas encontrados: {len(alerts)}") return alerts

===== FUNÇÃO: Filtrar por palavra-chave =====

def filtrar_alertas(alerts, keyword): return [a for a in alerts if keyword.lower() in a['alert'].lower()]

===== FUNÇÃO: Estatísticas resumidas =====

def estatisticas(alerts): resumo = {} for a in alerts: risco = a['risk'] resumo[risco] = resumo.get(risco, 0) + 1 return resumo

===== FUNÇÃO: Exportar CSV =====

def export_csv(alerts, filename="zap_alertas.csv"): keys = ['alert', 'risk', 'url', 'param', 'evidence'] with open(filename, 'w', newline='', encoding='utf-8') as f: writer = csv.DictWriter(f, fieldnames=keys) writer.writeheader() for a in alerts: writer.writerow({k: a.get(k, "") for k in keys}) print(f"[+] CSV exportado: {filename}")

===== FUNÇÃO: Exportar JSON =====

def export_json(alerts, filename="zap_alertas.json"): with open(filename, 'w', encoding='utf-8') as f: json.dump(alerts, f, indent=2, ensure_ascii=False) print(f"[+] JSON exportado: {filename}")

===== FUNÇÃO: Mostrar alertas filtrados =====

def mostrar_alertas_categoria(alerts, categoria): print(f"\n=== {categoria.upper()} ===") categoria_alerts = filtrar_alertas(alerts, categoria) if categoria_alerts: for a in categoria_alerts: print(f"- {a['alert']} (Risco: {a['risk']}) -> {a['url']}") else: print("Nenhum alerta encontrado nessa categoria.")

===== PROGRAMA PRINCIPAL =====

if *name* == "*main*": zap = connect_zap() spider_scan(zap, TARGET) passive_scan(zap)

active_scan(zap, TARGET)

alerts = get_alerts(zap, TARGET)

```
# Estatísticas gerais
stats = estatisticas(alerts)
print("\n=== Estatísticas por nível de risco ===")
for risco, count in stats.items():
    print(f"{risco}: {count} alertas")

# Mostrar alertas filtrados por categoria OWASP
mostrar_alertas_categoria(alerts, "Injection")
mostrar_alertas_categoria(alerts, "Cross Site Scripting")
mostrar_alertas_categoria(alerts, "Server Side Request Forgery")
# Broken Access Control
mostrar_alertas_categoria(alerts, "Access Control")

# Exportar relatórios
export_json(alerts, "zap_alertas_completo.json")
export_csv(alerts, "zap_alertas_completo.csv")

print("\n[+] Scan concluído com sucesso!")
```

""" zap_bac_advanced.py ⚠️ Execute apenas em ambiente de laboratório controlado.

Este script:

- Conecta ao ZAP
- Cria contexto e usuários
- Faz Spider, Passive Scan, Active Scan

- Coleta alertas de Injection, XSS, SSRF, BAC
- Compara respostas entre usuários para BAC
- Exporta relatórios JSON/CSV detalhados """

===== CONFIGURAÇÃO =====

ZAP_ADDRESS = '127.0.0.1' ZAP_PORT = '8080' TARGET = 'http://juice-shop.lab/'

API_KEY = None CONTEXT_NAME = "LabContext"

Usuários para Broken Access Control

USERS = [{"name": "normal", "username": "user", "password": "userpass"}, {"name": "admin", "username": "admin", "password": "adminpass"}]

===== FUNÇÃO: Conectar ao ZAP =====

def connect_zap(): try: zap = ZAPv2(apikey=API_KEY, proxies={'http': f'http://{ZAP_ADDRESS}:{ZAP_PORT}', 'https': f'http://{ZAP_ADDRESS}:{ZAP_PORT}'}) print(f"[+] Conectado ao ZAP versão {zap.core.version}") return zap except Exception as e: print(f"[ERRO] Conexão ZAP falhou: {e}") sys.exit(1)

===== FUNÇÃO: Criar contexto e usuários =====

def setup_context(zap): print(f"[+] Criando contexto '{CONTEXT_NAME}'") context_id = zap.context.new_context(CONTEXT_NAME) for user in USERS: user_id = zap.users.new_user(context_id, user["name"])
zap.users.set_authentication_credentials(context_id, user_id, f"username={user['username']}&password={user['password']}")
zap.users.set_user_enabled(context_id, user_id, True) print(f"[+] Contexto e usuários configurados") return context_id

===== FUNÇÃO: Spider =====

def spider_scan(zap, target): scan_id = zap.spider.scan(target) while int(zap.spider.status(scan_id)) < 100: print(f"Spider: {zap.spider.status(scan_id)}%") sleep(2) print("[+] Spider finalizado")

===== FUNÇÃO: Passive Scan =====

def passive_scan(zap): while int(zap.pscan.records_to_scan) > 0: print(f"Passive scan restante: {zap.pscan.records_to_scan}") sleep(2) print("[+] Passive Scan finalizado")

===== FUNÇÃO: Active Scan =====

def active_scan(zap, target): scan_id = zap.ascan.scan(target) while int(zap.ascan.status(scan_id)) < 100: print(f"Active scan: {zap.ascan.status(scan_id)}%") sleep(5) print("[+] Active Scan concluído")

===== FUNÇÃO: Comparar usuários para BAC =====

def analyze_bac(zap, context_id): print("[+] Analisando Broken Access Control por comparação de usuários") # O ZAP detecta falhas de autorização automaticamente se contexto/usuários configurados # Coletar alertas específicos de Access Control alerts = zap.core.alerts(baseurl=TARGET) bac_alerts = [a for a in alerts if "Access Control" in a['alert']] return bac_alerts

===== FUNÇÃO: Exportar JSON/CSV =====

def export_reports(alerts, json_file="alerts.json", csv_file="alerts.csv"): with open(json_file, 'w', encoding='utf-8') as f: json.dump(alerts, f, indent=2, ensure_ascii=False) with open(csv_file, 'w', newline='', encoding='utf-8') as f: keys = ['alert', 'risk', 'url', 'param', 'evidence'] writer = csv.DictWriter(f, fieldnames=keys) writer.writeheader() for a in alerts: writer.writerow({k: a.get(k, "") for k in keys}) print(f"[+] Relatórios exportados: {json_file}, {csv_file}")

===== FUNÇÃO: Estatísticas =====

def stats_by_risk(alerts): stats = {} for a in alerts: stats[a['risk']] = stats.get(a['risk'], 0) + 1 return stats

===== PROGRAMA PRINCIPAL =====

if *name* == "*main*": zap = connect_zap() context_id = setup_context(zap) spider_scan(zap, TARGET) passive_scan(zap) active_scan(zap, TARGET)

alerts = zap.core.alerts(baseurl=TARGET)

Estatísticas

```
stats = stats_by_risk(alerts)
print("\n=== Estatísticas por nível de risco ===")
for risco, count in stats.items():
    print(f"{risco}: {count} alertas")

# BAC avançado
bac_alerts = analyze_bac(zap, context_id)
print(f"\n=== Broken Access Control (comparação de usuários) ===")
if bac_alerts:
    for a in bac_alerts:
        print(f"- {a['alert']} (Risco: {a['risk']}) -> {a['url']}")
else:
    print("Nenhum alerta de BAC detectado.")

# Exportar relatórios
export_reports(alerts, "zap_full_advanced.json", "zap_full_advanced.csv")
print("\n[+] Scan completo e relatórios gerados!")
```