INSTITUTO POLITÉCNICO DE BEJA

Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia de Segurança Informática

Linguagens de Programação Dinâmicas

PROJETO PRÁTICO

Análise e Desenvolvimento de Ferramentas de Cibersegurança em Python

Rafael Conceição Narciso - 24473

Beja, janeiro de 2026

INSTITUTO POLITÉCNICO DE BEJA
Escola Superior de Tecnologia e Gestão
Mestrado em Engenharia de Segurança Informática
Linguagens de Programação Dinâmicas

PROJETO PRÁTICO

Relatório de Projeto Prático

Rafael Conceição Narciso - 24473

Docente Responsável: Armando Ventura

Beja, janeiro de 2026

Resumo

Este relatório documenta o desenvolvimento do *PySecurity Toolkit*, uma aplicação modular desenvolvida em Python que demonstra a versatilidade das linguagens de programação dinâmicas na Cibersegurança. O projeto integra funcionalidades ofensivas, como testes de carga de rede (DoS) e varrimento de portas, e defensivas, incluindo *Port Knocking*, gestão de credenciais com 2FA e comunicação segura. A implementação recorre a bibliotecas como Scapy, Socket, Threading e Cryptography, explorando a capacidade do Python de interagir com o baixo nível da rede e algoritmos criptográficos.

Palavras-chave: Python, Scapy, DoS, Criptografia, 2FA, Chat P2P.

Abstract

This report documents the development of the *PySecurity Toolkit*, a modular application built in Python aimed at demonstrating the versatility of dynamic programming languages in Cybersecurity. The project integrates offensive functionalities, such as network stress testing (DoS) and port scanning, as well as defensive features, including Port Knocking, secure credential management with 2FA, and encrypted communication. The implementation utilizes libraries such as Scapy, Socket, Threading, and Cryptography, exploring Python's ability to interact with low-level networking and complex cryptographic algorithms.

Keywords: Python, Scapy, DoS, Encryption, 2FA, P2P Chat.

Índice

Índice de Códigos

1 Introdução

O presente projeto consiste no desenvolvimento de uma *suite* modular de ferramentas de segurança ofensiva (*Red Teaming*) e defensiva (*Blue Teaming*), denominada **PySecurity Toolkit**. A escolha de **Python** justifica-se pela sua natureza dinâmica e interpretada. Ao contrário de linguagens como C, onde a gestão de memória e *buffers* é manual e propensa a erros (ex: *buffer overflows*), o Python oferece uma abstração segura e bibliotecas robustas para manipulação de *sockets* e protocolos de rede (**pythonDocs**).

2 Análise de Rede e Manipulação de Pacotes

A primeira fase de qualquer auditoria de segurança é o reconhecimento (*Recon*). Para tal, o projeto explora a interação com as camadas de Transporte e Rede do modelo OSI.

2.1 Port Scanning e o Handshake TCP

O varrimento de portas (*Port Scanning*) serve para identificar quais os serviços que estão ativos num servidor (ex: porta 80 indica um servidor Web, porta 22 indica SSH). Para este módulo, utilizou-se a biblioteca padrão `socket`. O script implementa um **TCP Connect Scan**, que se baseia no funcionamento padrão do protocolo TCP, conhecido como *Three-Way Handshake* :

1. **SYN:** O cliente envia um pedido de sincronização ("Posso entrar?").
2. **SYN-ACK:** Se a porta estiver aberta, o servidor aloca recursos e responde afirmativamente.
3. **ACK:** O cliente confirma a receção e a conexão é estabelecida.

No código desenvolvido, a função `socket.connect_ex((ip, port))` automatiza este processo. Se a função retornar `0`, significa que o *handshake* foi concluído com sucesso (Porta Aberta).

```python
import socket
def scan_port(ip, port):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(0.5)
        # Retorna 0 se o handshake for completado (Porta Aberta)
        result = sock.connect_ex((ip, port))
        if result == 0:
            print(f"[+] Porta {port} ABERTA")
        sock.close()
    except Exception as e:
        print(f"Erro: {e}")
```

Código 1: Port Scanner com Sockets

3 Ataques de Negação de Serviço (DoS)

Para simular ataques de stress, foi necessário utilizar *Raw Sockets* via biblioteca **Scapy** (**scapyDocs**) para contornar as regras do Sistema Operativo.

3.1 Ataque SYN Flood (Camada 4)

O SYN Flood é um ataque que visa a **exaustão de memória** do servidor. Ele explora o facto de o servidor ter de "reservar lugar"para cada nova conexão na sua Fila de Espera (*Backlog Queue*) .

- **Teoria (Half-Open Connections):** O atacante envia milhares de pacotes SYN, mas **nunca** envia o ACK final. O servidor mantém a conexão "meio-aberta"em memória à espera da resposta. Quando a memória enche, o servidor

rejeita conexões legítimas.

- **Implementação:** O uso do Scapy é obrigatório para enviar pacotes sem completar o handshake. Implementou-se também **IP Spoofing** (`RandIP()`) para ocultar a origem e evitar que a própria máquina do atacante envie um pacote RST (Reset) que anularia o ataque.

```
from scapy.all import IP, TCP, send, RandIP, RandShort
def syn_flood(target_ip, target_port, count=1000):
    for _ in range(count):
        # 1. IP Spoofing: Origem Aleatoria para evitar RST
        # 2. Flag SYN ativa para iniciar conexao
        packet = IP(src=RandIP(), dst=target_ip) / \
                 TCP(sport=RandShort(), dport=target_port, flags="S
    ")
        send(packet, verbose=0)
```

Código 2: SYN Flood com Scapy e IP Spoofing

3.2 Ataque UDP Flood

O protocolo UDP é *connectionless*. O ataque UDP Flood visa saturar a largura de banda ou a CPU do alvo.

- **Mecanismo de Exaustão:** O script envia datagramas UDP com dados aleatórios para portas fechadas. O servidor é obrigado a processar o pacote e gerar uma resposta ICMP "Destination Unreachable", consumindo ciclos de CPU intensivos.

4 Mecanismos de Defesa e Ocultação

4.1 Port Knocking

O *Port Knocking* implementa "Segurança por Obscuridade". A firewall bloqueia todas as conexões, exceto se detetar uma "assinatura"específica: uma sequência de pacotes SYN em portas pré-definidas (ex: 7000 $\rightarrow$ 8000 $\rightarrow$ 9000). O script Python automatiza o envio desta sequência, instruindo o servidor a abrir dinamicamente a porta 22 (SSH).

5 Criptografia e Comunicação Segura

5.1 Gestão de Credenciais (Fernet e 2FA)

O módulo de *Password Manager* utiliza a especificação **Fernet** (AES-128 em modo CBC com assinatura HMAC) para garantir confidencialidade e integridade (**cryptographyLib**). Adicionalmente, implementou-se autenticação TOTP (RFC 6238), onde o código é gerado matematicamente com base no tempo atual (T) e numa chave secreta (K):

$$TOTP = \text{Truncate}(HMAC\text{-}SHA1(K, \lfloor \frac{T - T_0}{30} \rfloor))$$

6 Comunicação em Tempo Real: Chat P2P

Para demonstrar a capacidade de programação de redes avançada em Python, desenvolveu-se um sistema de Chat Peer-to-Peer (P2P) encriptado.

6.1 Fundamentação Teórica: Concorrência e Sockets

Numa comunicação de rede TCP tradicional (*single-thread*), as operações de entrada e saída (I/O) são bloqueantes.

- **O Problema do Bloqueio:** A função `socket.recv()` suspende a execução do programa até que cheguem dados pela rede. Enquanto o programa espera por uma mensagem, o utilizador fica impossibilitado de escrever ou enviar dados, resultando numa experiência de "Walkie-Talkie"(apenas um fala de cada vez) em vez de um Chat real.

- **A Solução (Multithreading):** Para permitir uma comunicação *Full-Duplex* (envio e receção simultâneos), utilizou-se a biblioteca `threading` (**threadingDocs**). O programa divide-se em dois fluxos de execução paralelos :

  1. **Main Thread (Envio):** Gere a interface com o utilizador, lê o *input* do teclado, encripta a mensagem com AES e envia para o socket.
  2. **Listener Thread (Receção):** Corre em segundo plano (modo *Daemon*). Fica num loop infinito bloqueado no `recv()`. Assim que chegam dados, desencripta-os e imprime-os no ecrã imediatamente, sem interromper o utilizador que está a escrever.

6.2 Implementação Prática

O código abaixo demonstra a criação da *thread* de escuta e o processo de encriptação antes do envio.

```
1  import threading
2  import socket
3  from cryptography.fernet import Fernet
4
5  def receive_messages(sock, cipher):
6      """Funcao que corre na Thread Secundaria"""
7      while True:
8          try:
9              # recv() bloqueia aqui, mas nao para o programa
       principal
10             data = sock.recv(1024)
```

```
            if not data: break

            # Desencripta a mensagem recebida
            msg = cipher.decrypt(data).decode()
            print(f"\n[Amigo]: {msg}")
        except:
            break

def start_chat(sock, key):
    cipher = Fernet(key)

    # Inicia a Thread de escuta (Daemon = fecha quando o programa fechar)
    listener = threading.Thread(target=receive_messages, args=(sock, cipher))
    listener.daemon = True
    listener.start()

    # Thread Principal: Trata do envio
    while True:
        msg = input("Eu: ")
        encrypted_msg = cipher.encrypt(msg.encode())
        sock.send(encrypted_msg)
```

Código 3: Implementação do Chat com Threading e AES

7 Conclusão

O desenvolvimento do *PySecurity Toolkit* permitiu validar a eficácia do Python em Cibersegurança. A combinação de *raw sockets* (ataques), criptografia robusta (defesa) e concorrência (comunicação) demonstra que as linguagens dinâmicas, apesar da gestão automática de memória, oferecem o controlo de baixo nível necessário para ferramentas de segurança profissionais.