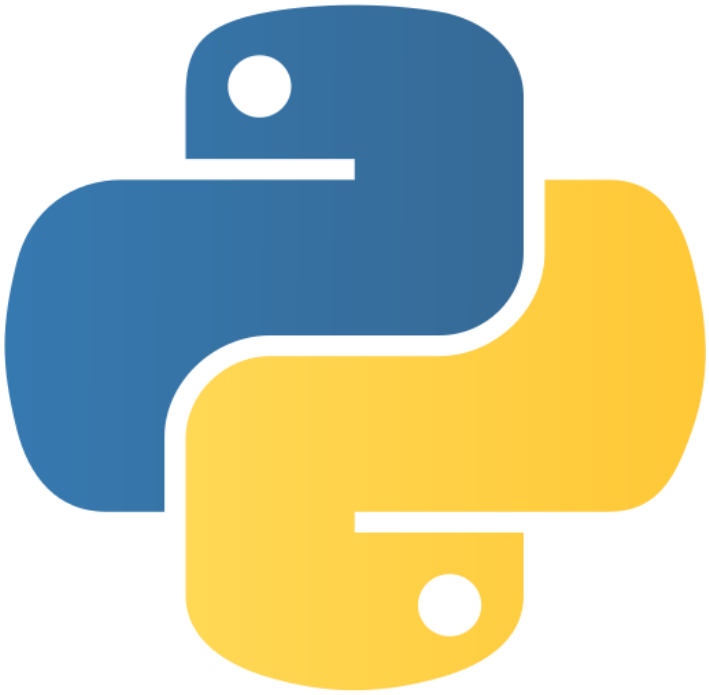
Relatório CP3, Security Coding:

Sniffer JSON CLI



Relatório destinado ao processo de criação da CP3, que retrata uma ferramente que tem a capacidade de analisa o tráfego da rede e criar um arquivo json com base nos dados obtidos para a análise, desenvolvido em python.

Professor: Marcio Cruz

RM568040 – Lucas Otavio Lacerda Menezes

RM567681 – Guilherme Augusto Figueiredo Tesser

*I*dealização:

Tema Escolhido: Capturador e Formatador de Tráfego de Rede (Sniffer JSON CLI).

Nome da Ferramenta: sniffer_json.py

Objetivo: Ser uma ferramenta CLI que automatiza a criação de arquivo json e deixa em um formato que facilita a análise.

Abordagem Selecionadas:

- **Bibliotecas obrigatórias:** argparse, datetime, logging.
- **Bibliotecas não obrigatórias: json, scapy**
- **PEP8**

Entrada esperada:
py sniffer_json.py --interface eth0 --output saida.json --count 500 --filter "tcp port 80"

Estrutura esperada de saída (JSON) (EXEMPLO GERADO POR IA):


```
[
 {
  "timestamp": "2025-10-29 16:15:30.123456",
  "protocol": "TCP",
  "source_ip": "192.168.1.100",
  "source_port": 54321,
  "destination_ip": "172.217.16.78",
  "destination_port": 443,
  "payload_length": 66,
  "flags": "SYN, ACK"
 }
]
```


Esta estrutura reflete os dados cruciais obtidos no tratamento de cada pacote. Notamos que nem todo pacote será TCP. Em casos de pacotes UDP, o campo `flags` será, objetivamente, N/A, já que o UDP não possui mecanismos de controle de conexão como o TCP.

Pré-Desenvolvimento:

Antes de colocar a mão na massa analisamos as possibilidades e fizemos projeções mentais do resultado para saber que caminho seguir, partes antes dessa seção do relatório definem nosso objetivo e servem como guia para a organização do desenvolvimento e estudos a serem feitos antes de iniciar o “coding”.

Por ser feito em dupla, escolhemos a abordagem chamada pep 8, que consiste em padronizar o desenvolvimento em Python, além disso, com o tema em mente procuramos bibliotecas que nós auxiliariam a chegar ao nosso objetivo, sendo escolhidas a *json* para transformar os dicionários em arquivos .json, e a biblioteca *Scapy* para capturar os pacotes e transformá-los em dados usáveis, não seguimos com a biblioteca *PyShark* por mais interassante que seja, pois ela necessita da instalação e configuração do *TShark* o que adiciona uma camada de dependência não-python desnecessária, o *Scapy* é um *FrameWork* nativo do Python que ja cumpri premissa necessária.

A biblioteca *argparse* vai ser utilizada para criação de “flags”, que seriam opção para orientar a ferramenta, a *datatime* registraria o tempo instante quando o pacote foi capturado para criar um *timestamp* para facilidade em análise e criação de gráficos, e o logging para auxiliar o usuário.

Conforme a entrada esperada já citada, (python sniffer_json.py --interface eth0 --output saída.json --count 500 --filter "tcp port 80"), os argumentos definidos pelo argparse são:

--interface ou -i (qual placa utilizada para captura)

--output ou -o (onde o JSON será escrito)

--count ou -c (quantidade de pacotes desejado a serem capturados)

--filter ou -f (critérios de filtro de captura como protocolo ou porta).

Outra biblioteca utilizada foi a os para a criação dos arquivos de texto.

Lógica:

A lógica que usaríamos para fazer o código consiste em utilizar o argparse para puxar informações e coletar pacotes, a princípio vai existir uma função que captura um pacote por vez, quando ele captura esse pacote, ele passa por um tratamento, então esse pacote entra dentro de outra função, na onde extrai dados do pacote e organiza em um dicionário, com o dicionário feito, ele vai ser usado como entrada para outra função que vai transformar esse dicionário em um json e jogar ele pra dentro de uma lista, após todos os pacotes serem transformados em json e jogados para a lista que vai conter todos os jsons, vai executar outra função que decreta o fim do script que abre um arquivo em modo write e joga la a lista que tem todos os jsons.

Detalhamento da Implementação:

A seção "Desenvolvimento" do relatório descreve uma lógica processual de capturar um pacote, tratá-lo e adicioná-lo a uma lista. Na prática, a implementação final utilizou uma abordagem mais eficiente e padrão da biblioteca Scapy, baseada em callbacks.
A função sniff da Scapy foi utilizada com três argumentos principais que definem o núcleo da nossa ferramenta:

- prn=listar_json: Este é o argumento mais crítico. Em vez de criarmos um loop manual para capturar pacote por pacote, instruímos a função sniff a executar automaticamente a função listar_json para cada pacote que ela captura. Isso é um callback.
- store=False: Conforme o código, foi definido que os pacotes não deveriam ser armazenados na memória pela própria Scapy. Isso é vital para capturas longas ou ilimitadas (quando count=0), pois evita o esgotamento da memória RAM. Nossa ferramenta gerencia os dados extraídos, não os pacotes brutos.
- count=args.count: Este argumento permite que a função sniff pare automaticamente após o número de pacotes especificado, ou continue indefinidamente (se 0) até que o usuário interrompa.

A lógica de armazenamento também foi refinada. Em vez de transformar cada dicionário em um JSON individualmente e depois adicionar a uma lista, o código adota uma abordagem mais performática:
A função listar_json (o callback) chama construcao_dicio para obter um dicionário Python. Este dicionário é anexado diretamente à variável global lista_com_jsons.
Somente ao final da execução (após o sniff terminar ou ser interrompido), a função salvar_arquivo é chamada. Ela utiliza json.dump() para converter a lista inteira de dicionários em um único arquivo JSON formatado.

Estrutura das Funções:

O código foi modularizado seguindo boas práticas e o padrão PEP8:

- **construcao_dicio(pacote):** Traduz o objeto de pacote da Scapy em um dicionário simples com os campos relevantes das camadas IP, TCP e UDP. Para pacotes não IP, registra um resumo para não perder o evento.
    - O campo originalmente planejado como *payload_length* foi renomeado para *cabecalho*.
- **listar_json(pacote):** Callback de processamento. Chama *construcao_dicio*, adiciona o resultado à lista global e registra a ação no log.
- **salvar_arquivo(caminho_arquivo):** Garante que o diretório de saída exista e grava toda a lista de pacotes em um arquivo JSON usando *json.dump()*.
- **main():** É o ponto de entrada. Configura o *argparse*, inicia o logger, chama *sniff* e gerência erros e finalização do script.

Tratamento de Erros:

Durante o desenvolvimento, foram implementados mecanismos para aumentar a robustez e garantir que a ferramenta funcione de forma estável e amigável ao usuário.

- **Interrupção do Usuário (KeyboardInterrupt):**
  A chamada da função *sniff* foi encapsulada em um bloco *try...except*. Quando o usuário interrompe a execução com *Ctrl+C* (em capturas contínuas, por exemplo), o programa não é encerrado de forma abrupta. Em vez disso, exibe uma mensagem informando que os dados estão sendo salvos e chama *salvar_arquivo* antes de finalizar.
- **Erro de Permissão (PermissionError):**
  Como a captura de pacotes normalmente requer privilégios de administrador, o código trata antecipadamente esse erro. Caso ocorra, é exibida uma mensagem explicativa (“Erro: Permissão negada...”), em vez de um erro genérico do Python.
- **Gerenciamento de Caminhos (pathlib):**
  O uso de *pathlib* garante que o diretório de destino seja criado automaticamente, evitando falhas por diretórios inexistentes e tornando o uso mais prático.
- **Logging (logging):**
  A biblioteca *logging* foi utilizada para registrar o funcionamento da ferramenta em tempo real. O usuário é informado sobre o início da execução, o processamento de pacotes, a gravação do arquivo final e possíveis erros, oferecendo transparência e controle durante o uso.

Conclusão do Desenvolvimento:

O projeto **sniffer_json.py** atingiu com sucesso o objetivo de criar uma ferramenta de linha de comando capaz de capturar tráfego de rede e formatá-lo em JSON para análise.

As bibliotecas obrigatórias (*argparse*, *datetime*, *logging*) e as adicionais (*json*, *scapy*) foram integradas de acordo com o padrão **PEP8**, garantindo clareza e organização no código.

O principal aprimoramento em relação ao plano inicial foi a adoção de uma **arquitetura baseada em callback** (*prn=listar_json*), que se mostrou mais eficiente e elegante do que um processamento em loop manual.

A ferramenta final apresentou também **boa robustez**, tratando interrupções do usuário e erros de permissão de forma controlada, assegurando uma execução estável e confiável.

Utilização da IA:

Objetivamente a IA só vai ser utilizada para fomentação de ideias e construção de logicas e auxílio para pesquisas, a princípio o relatório e código não terá sequer uma linha escrita por IA (além de exemplos).

Prompts utilizados:

- “como eu usaria o logging? a biblioteca”
- “daria pra usar o dvwa para fazer experiencias?"
- “gere um exemplo de como seria o arquivo json esperado ao ser gerado pela ferramenta”
- “me fale como organizar o documento de inicio antes de começar a fazer tudo, vou documentar enquanto a gnt cria o script, quero ter uma nocao de organização”
- “socket é oq? daria pra usar o pyshark e o scapy qual o melhor”
- “oq seria timestamp”
- “import scapy n vai funcionar? porque usa from scapy.all import * e o from datetime import datetime”
- “oq seria callback me de exemplos ilustrativos como se fosse uma criança use analogia”
- “oq é a função lambda”

Fontes:

https://awari.com.br/aprenda-a-capturar-e-analisar-pacotes-de-rede-com-python/

https://docs.python.org/3/library/argparse.html

https://docs.python.org/3/library/json.html

https://docs.python.org/3/library/logging.html

https://docs.python.org/3/library/datetime.html

https://docs.python.org/3/library/pathlib.html

https://scapy.readthedocs.io/

Nota: no final usei mt IA pra fazer o relatório por falta de tempo :/