Anomaly Detection System Using ComproAlert

Introduction

The provided code implements a comprehensive system for aggregating port-level network traffic statistics and detecting anomalies using an autoencoder-based approach. The system utilizes an eBPF program to parse network packets, aggregate traffic statistics for TCP and UDP flows, and employs a Python script for training an autoencoder and detecting anomalies in the aggregated data.

Components

1. eBPF Program (port_flow.c)

The eBPF program parses incoming network packets and aggregates traffic statistics for TCP and UDP flows. It maintains a hash table (flow_data) to store per-port flow statistics, including total bytes sent, total packets, average payload size, maximum payload size, and minimum payload size.

2. Python Script (port_flow_aggregation_anomaly_detection.py)

The Python script uses BCC to load and attach the eBPF program, collects flow statistics, trains an autoencoder, and detects anomalies in the aggregated port traffic data. Key functionalities include initializing and training an autoencoder, periodically collecting flow statistics, standardizing and training the autoencoder, and detecting anomalies with subsequent alerts.

3. Configuration (config.py)

input_dim: Dimension of the input data for the autoencoder.

encoding_dim: Dimension of the encoded data in the autoencoder.

prediction_period: Interval for collecting port traffic statistics and anomaly detection.

training_period: Interval for training the autoencoder.

Autoencoder-based Anomaly Detection

Autoencoder Initialization

The script initializes an autoencoder model with a specified input and encoding dimension. The autoencoder is a neural network with a single hidden layer using the Mean Squared Error loss function.

Training the Autoencoder

The script periodically trains the autoencoder using the aggregated port traffic data. Training involves standardizing the dataset and fitting it to the autoencoder model, and it is performed for a fixed number of epochs.

Anomaly Detection

Anomalies are detected by comparing the Mean Squared Error (MSE) between the original and reconstructed data. A threshold is set based on the 95th percentile of the MSE during training. If the MSE exceeds the threshold, the data point is considered an anomaly.

Execution

The script runs in an infinite loop, periodically collecting and processing port traffic statistics. Autoencoder training and anomaly detection occur at specified intervals, triggering alerts for anomalies sent to a specified endpoint via an HTTP POST request.

Dependencies

The script relies on BCC, NumPy, requests, and Keras. Ensure these dependencies are installed.

Conclusion

This system demonstrates the use of eBPF for real-time traffic aggregation and an autoencoder for anomaly detection in network traffic. Users can adapt and extend the code for specific use cases, such as network monitoring and anomaly detection in port-level traffic.

IP Address Collection and Geolocation-Based Filtering System

Introduction

The provided code implements a system for collecting IP addresses using an eBPF program and performing geolocation-based filtering using the HDBSCAN clustering algorithm. The system aims to detect anomalies or outliers in the collected IP addresses based on their geolocation information.

Components

1. eBPF Program (ipaddress_collection.c)

The eBPF program collects IP addresses from incoming network packets. It is attached to a raw socket, filters packets based on their Ethernet and IP headers, and stores the source IP addresses in a BPF hash table (src_ips).

2. Python Script (geolocation_filtering.py)

The Python script utilizes the BCC library to load and attach the eBPF program and integrates with the HDBSCAN clustering algorithm for geolocation-based filtering. Key functionalities include loading the eBPF program, initializing and updating an HDBSCAN model, periodically checking for outliers, and sending alerts for detected outliers.

3. Python Script Configuration (config.py)

update_interval_seconds: Interval for updating the HDBSCAN model.

prediction_period: Interval for collecting IP addresses and performing geolocation-based filtering.

Geolocation-Based Filtering

IP to Geolocation Conversion

The script converts IP addresses to dotted-decimal format using the int_to_dotted_decimal function. It queries an external API (http://ip-

api.com/json/) to obtain geolocation information for each IP address using the send_get_request function.

HDBSCAN Model

The script initializes and updates an HDBSCAN model using the collected geolocation data. It prints cluster centers and data statistics after model updates.

Outlier Detection

The script detects outliers by comparing the distance of each geolocation point to the centroids of HDBSCAN clusters. Outliers trigger alerts and are sent to a specified endpoint using an HTTP POST request.

Execution

The script runs in an infinite loop, periodically updating the HDBSCAN model and performing geolocation-based filtering. KeyboardInterrupt (Ctrl+C) can be used to stop the script.

Dependencies

The script relies on Python, BCC, HDBSCAN, NumPy, and requests. Ensure these dependencies are installed.

Conclusion

This system provides a practical example of using eBPF for IP address collection and HDBSCAN for geolocation-based filtering to detect outliers. Users can adapt and extend the code for specific use cases, such as network anomaly detection and security monitoring.

Ransomware Detection With Process Behaviour Analysis

Section 0: Process Behavior Analysis for Ransomware Detection

1. System Call Monitoring (section 1)

The first script continuously monitors specific system calls (e.g., read, write, openat, unlink, and rename) made by processes on a Linux system using eBPF.

2. Data Collection (section 1)

Data on the frequency of monitored system calls for each process is collected, providing insights into process behaviour.

3. Anomaly Detection with Autoencoder (section 2)

The second script includes SystemCallAutoencoder, which is used for anomaly detection. Data is preprocessed, and an autoencoder model is trained to learn normal process behaviour.

4. Anomaly Identification using Autoencoder (section 2)

During real-time operation, the script calculates Reconstruction Mean Squared Errors (MSE) for system call data points. If the calculated MSE exceeds a predefined threshold, it indicates anomalous behaviour.

5. Ransomware Detection

Ransomware often exhibits unusual behaviour, which can trigger anomalies. Anomalies in system call patterns may indicate ransomware execution, prompting actions such as process isolation or system alerts.

By monitoring and analysing system call patterns, these scripts assist in detecting processes that deviate from normal behaviour, potentially signalling ransomware or other malicious activities. Proper threshold configuration and response actions are crucial for effective ransomware detection.

Section 1: System Call Monitoring with eBPF

Platform

Linux, eBPF environment

Dependencies

bcc (BPF Compiler Collection) library

Description

This section provides an overview and documentation of a Python script that tilizes Ebpf (extended Berkeley Packet Filter) to monitor specific system call activities on a Linux system. The script tracks and records the number of occurrences of five different system calls (read, write, openat, unlink, and rename) for various processes and identifies the process with the maximum count for the "openat" system call. This code is designed to run in a Linux environment with eBPF support.

Code Overview

The provided Python code defines a Probe class that creates and manages an Ebpf program to trace and count specific system calls. The program uses Ebpf hash tables to store and update counts for each system call. It also includes a method to retrieve the maximum counts for each tracked system call.

Initialization

The Probe class initializes the Ebpf program text, which includes the necessary headers and definitions, as well as the tracepoint probes for the specified system calls (read, write, openat, unlink, and rename).

Tracepoint Probes

The code defines tracepoint probes for each of the monitored system calls (sys_enter_read, sys_enter_write, sys_enter_openat, sys_enter_unlink, and sys_enter_rename).

Inside each probe, it extracts the process ID (PID) and updates the corresponding system call hash table's entry with the count of the specific system call.

Retrieving Maximum Values

The get_max_values method retrieves the maximum counts for each of the monitored system calls.

It iterates through the hash tables associated with each system call, identifies the process with the maximum count for the "openat" system call, and uses that process to extract the counts for "read" and "write" system calls. Similarly, it finds the maximum count for the "unlink" and "rename" system calls.

The maximum counts are stored in a dictionary called max_values, which is returned as the result.

The process IDs corresponding to the system call counts stored in 'max_pids' are also returned.

Section 2: Real-time Anomaly Detection with System Call Autoencoder

Platform

Linux, eBPF environment

Dependencies

Python (version 3.10 or above) with the following modules:

sklearn

Tensorflow (>= 2.0)

Description

This section provides an overview and documentation of a Python script that implements real-time anomaly detection using a custom SystemCallAutoencoder class. The script collects data on system call activities and trains an autoencoder model to detect anomalies in real-time. The code utilises the Probe class from a previous code snippet, which uses eBPF to monitor system call counts on a Linux system.

Code Overview

The Python script is designed to perform the following tasks:

Initialise a dataset for system call activities.

Define and create a SystemCallAutoencoder class for anomaly detection.

Specify time intervals for data collection, model training, and prediction using the config.py file.

Train the autoencoder model with collected data.

Continuously collect system call data, train the model periodically, and predict anomalies.

System Call Autoencoder Class

The System Call Autoencoder class is responsible for creating and training an autoencoder model for anomaly detection. Below are the key components of this class:

Initialization (__init__):

Initialises the input and hidden dimensions for the autoencoder model.

Creates the autoencoder model using TensorFlow, with linear activation functions.

Compiles the model using the Adam optimizer and mean squared error (MSE) loss function.

Training (train_autoencoder):

Scales the input data using the StandardScaler.

Fits the scaler to the dataset and transforms the dataset.

Trains the autoencoder model with the scaled dataset for a specified number of epochs and batch size.

Prediction (predict_anomalies):

Predicts anomalies for a given data point by comparing the reconstructed data point with the original data point.

Calculates the mean squared error (MSE) between the scaled input data and the reconstructed data point.

Compares the MSE to a predefined threshold to determine if an anomaly is detected.

MSE Calculation (calculate_mse):

Calculates the MSEs for a dataset using the trained autoencoder model.

Main Execution

The main execution section of the script performs the following tasks:

Initializes the dataset and the System Call Autoencoder instance.

Loads configurations from the config.py file.

Loops indefinitely to collect data, train the model, and predict anomalies in real-time.

Collects the latest system call data using the Probe class.

Trains the autoencoder model with the collected data at specified training intervals and calculates the mean and standard deviation of the reconstruction MSEs to find the threshold, which is equal to the mean + 3 * standard deviation.

Predicts anomalies based on the trained model and predefined threshold. If an anomaly is detected, the process IDs corresponding to the system call counts are displayed.

Configurations

To modify the data_collection_interval and training_interval, use the config.py file. Open the file and adjust the values accordingly.

Code Execution

To use this code, follow these steps:

Open a terminal window and switch to superuser/root.

Run the script (which continuously collects data, trains the model, and predicts anomalies in real-time) in a manner similar to the following:

bash

sudo python anomaly_detection.py

Press Ctrl + Z to stop execution.

DNS Monitoring Script

This script is designed for monitoring DNS traffic and performing anomaly detection on domain names using a combination of machine learning and n-gram models. It leverages eBPF (extended Berkeley Packet Filter) for efficient packet filtering and processing.

Prerequisites

Python Libraries:

requests: For making HTTP requests.

pickle: For serializing and deserializing Python objects.

keras: For building and training the autoencoder model.

tldextract: For extracting top-level domains from URLs.

nltk: For natural language processing tasks.

numpy: For numerical operations.

sklearn.preprocessing.MinMaxScaler: For scaling data.

dnslib: A library for parsing and building DNS packets.

eBPF Probes

The script contains eBPF probes attached to UDP (trace_udp_sendmsg) and TCP (trace_tcp_sendmsg) sendmsg functions. These probes capture DNS-related network traffic and extract information about the process, such as PID, UID, GID, and command name. The information is stored in an eBPF table named proc_ports.

DNS Monitoring Script

Overview

The DNS Monitoring Script is designed for monitoring DNS traffic and performing anomaly detection on domain names using a combination of machine learning and n-gram models. The script leverages eBPF (extended Berkeley Packet Filter) for efficient packet filtering and processing. It includes eBPF probes, an anomaly detection model, and functions for checking DNS requests against whitelists, blacklists, and machine learning models.

Prerequisites

Python Libraries

requests: For making HTTP requests.

pickle: For serializing and deserializing Python objects.

keras: For building and training the autoencoder model.

tldextract: For extracting top-level domains from URLs.

nltk: For natural language processing tasks.

numpy: For numerical operations.

sklearn.preprocessing.MinMaxScaler: For scaling data.

dnslib: A library for parsing and building DNS packets.

eBPF Probes

The script contains eBPF probes attached to UDP (trace_udp_sendmsg) and TCP (trace_tcp_sendmsg) sendmsg functions. These probes capture DNS-related network traffic and extract information about the process, such as PID, UID, GID, and command name. The information is stored in an eBPF table named proc_ports.

DNS Matching eBPF Program

An eBPF program, dns_matching, is used to filter DNS packets from the network traffic. It looks for DNS-related packets (UDP or TCP) and checks for corresponding process information in the proc_ports table. If a match is found, it sends the information to user space.

Machine Learning Model

The script uses a pre-trained machine learning model (Domain_Model_Saved.h5) to predict anomalies in domain names based on their similarity to known domains. It also utilizes an n-gram model for cross-verification.

Domain Name Checking Functions

Functions (check_domain_name_list_combo and check_domain_name_list_n_gram) are provided for checking domain names against whitelists, blacklists, and machine learning models. These functions perform entropy checks on DNS requests and send alerts for rejected domain names.

Autoencoder for DNS Request Checking

The dns_req_checker instance of the Autoencoder class is specifically designed to detect anomalies in the entropy and length of DNS packets. This autoencoder is trained on DNS request properties to learn normal patterns and identify deviations that may indicate suspicious or malicious activity.

Autoencoder Class

The Autoencoder class is initialized with the input dimensions (input_dim) representing entropy and length. The encoding dimension (encoding_dim) defines the compressed representation of the input data. The autoencoder model is trained to reconstruct the input data

and, during this process, learns to distinguish normal patterns from anomalies.

Anomaly Detection

The train method of the Autoencoder class scales the input data and trains the model. The threshold for anomaly detection is set based on the 95th percentile of errors in the reconstruction process. The is_anomaly method then checks incoming DNS request properties against this threshold to identify anomalies.

DNS Request Properties

The DNS request properties include entropy (calculated using Shannon entropy formula) and the length of the DNS packet. Anomalies in these properties may indicate irregularities in DNS traffic, triggering alerts and further investigation.

Main Loop

The main loop continuously processes DNS-related network events captured by eBPF probes. It prints the status of accepted and rejected domain names based on various checks and alerts. The script periodically retrains the DNS request checker using the collected dataset.

Handling Interrupts

The script gracefully handles interrupts (Ctrl-C), prints the list of accepted and rejected domain names, and sends an alert email containing rejected domain names.

Configuration

Adjust the server address and port in the script to match your environment.

Customize the training and prediction periods in the script based on your requirements.

How to Run

Install the required Python libraries: pip install requests keras tldextract nltk numpy scikit-learn dnslib.

Ensure the eBPF dependencies are installed on the system.

Run the script: python DNS_monitoring.py.

Alerts

Rejected domain names are sent as alerts to the configured server. Additionally, rejected domain names are printed, and an email is sent with the list of rejected domain names.

Whitelist

Whitelisted domain names are fetched from the server and checked against incoming DNS requests.

Anomalies

Anomalies in DNS request properties are detected using an autoencoder model. If a domain name is suspected, it is cross-verified using an n-gram model.

Acknowledgments

The script utilizes eBPF technology and machine learning models for efficient DNS monitoring and anomaly detection. The n-gram model is used for cross-verifying domain name similarities.