Hands-On Manual: Data Collection & Vectorization Pipeline for SYSMON Logs

FOUNDATIONAL SETUP (All Teams Must Complete)

System Setup & Data Pipeline Question:

Goal: Establish a reproducible data collection and preparation pipeline using SYSMON utility, and Python.

Tasks:

1. **SYSMON Installation & Configuration**:
   - Install SYSMON on Windows host (use Sysmon64.exe with SwiftOnSecurity config)
   - **Configure event collection for:** Process Creation (ID 1), Network Connection (ID 3), File Creation (ID 11), Registry Events (ID 12-14)
   - Command: `Sysmon64.exe -accepteula -i sysmonconfig.xml`
2. **Log Collection & Parsing**:
   - **Export logs using PowerShell:** `Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational" | Export-Csv`
   - **Parse XML structure to extract:** ProcessId, CommandLine, Image, User, NetworkDestination, Hash
   - Create structured dataset with timestamp, event_code, and extracted fields
3. **Vectorization Pipeline**:
   - **Text preprocessing:** Tokenize CommandLine and Image paths, remove noise
   - **Choose embedding model:** sentence-transformers/all-MiniLM-L6-v2 OR OpenAI text-embedding-3-small
   - Generate embeddings for textual fields (CommandLine, ParentImage, etc.)
   - **Combine with numerical features:** event_code, ProcessId (one-hot encoded)
4. **Vector Database Storage**:
   - Install Chroma/Pinecone/Weaviate for vector storage
   - **Schema design:** `{event_id, timestamp, embedding_vector, metadata{event_code, label}}`
   - Implement batch insertion (1000 events/batch)
   - Create index for similarity search on embedding field
5. **Labeling Strategy**:
   - **Benign:** Normal system operations during controlled period (collected using SYSMON utility)
   - **Malicious:** Simulate simple attacks using system utilities (based on MITRE ATT&CK techniques) OR use public datasets (LANL, OpTC)
   - Map events to MITRE ATT&CK tactics (refer '**mitre-enterprise-attack-v18.1.xlsx**' file) using ATT&CK Navigator

Table of Contents

Prerequisites

Hardware Requirements

- Windows 10/11 or Windows Server (Administrator access required)
- Minimum 8GB RAM, 50GB free disk space
- Python 3.8+ installed

Software Requirements (Following to be executed on Command-Prompt (for Windows) or Terminal (for Linux/Mac))

```
# Create project directory
mkdir CTI_Pipeline
cd CTI_Pipeline

# Create Python virtual environment
python -m venv venv
# For Windows
venv\Scripts\activate
# For Linux/Mac
source venv/bin/activate

# Install required packages

pip install pandas numpy sentence-transformers chromadb openai scikit-
learn matplotlib seaborn
```

Download Required Tools

1. **SYSMON**: Download from Microsoft Sysinternals
2. **SwiftOnSecurity Config**: Download from GitHub

Part 1: SYSMON Installation & Configuration

Step 1.1: Download and Extract SYSMON (Following to be executed on a PowerShell terminal in Windows)

```
# Open PowerShell as Administrator
cd C:\CTI_Pipeline
```

```
# Download Sysmon
Invoke-WebRequest -Uri
"https://download.sysinternals.com/files/Sysmon.zip" -OutFile
"Sysmon.zip"

Expand-Archive -Path Sysmon.zip -DestinationPath .\Sysmon

# Download SwiftOnSecurity configuration
Invoke-WebRequest -Uri
"https://raw.githubusercontent.com/SwiftOnSecurity/sysmon-
config/master/sysmonconfig-export.xml" -OutFile "sysmonconfig.xml"
```

Step 1.2: Configure SYSMON for Target Events

# Create a custom configuration file sysmonconfig-custom.xml:

```
<Sysmon schemaversion="4.90">
  <EventFiltering>
    <!—Event ID 1: Process Creation →
    <ProcessCreate onmatch="include">
      <Image condition="begin with">C:\</Image>
    </ProcessCreate>

    <!—Event ID 3: Network Connection →
    <NetworkConnect onmatch="include">
      <Image condition="begin with">C:\</Image>
    </NetworkConnect>

    <!—Event ID 11: File Creation →
    <FileCreate onmatch="include">
      <TargetFilename condition="contains">\Users\</TargetFilename>
      <TargetFilename condition="contains">\Temp\</TargetFilename>
    </FileCreate>

    <!—Event ID 12-14: Registry Events →
    <RegistryEvent onmatch="include">
      <TargetObject
condition="contains">CurrentVersion\Run</TargetObject>
      <TargetObject
condition="contains">CurrentVersion\Windows\Run</TargetObject>
    </RegistryEvent>
  </EventFiltering>

</Sysmon>
```

Step 1.3: Install SYSMON (Following to be executed on a PowerShell terminal in Windows)

```
# Navigate to Sysmon directory
cd C:\CTI_Pipeline\Sysmon

# Install with custom configuration
.\Sysmon64.exe -accepteula -I ..\sysmonconfig-custom.xml

# Verify installation
Get-Service Sysmon64
```

```
# Check event log
Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational" -MaxEvents
10
```

Expected Output:

```
Status   Name               DisplayName
------   ----               -----------
Running  Sysmon64           Sysmon64
```

Step 1.4: Verification Script

# Create verify_sysmon.ps1:

```
# Check if Sysmon is running
$service = Get-Service Sysmon64 -ErrorAction SilentlyContinue
if ($service.Status -eq "Running") {
    Write-Host "Sysmon is running" -ForegroundColor Green
} else {
    Write-Host "Sysmon is not running" -ForegroundColor Red
}

# Count events by type
$eventCounts = @{
    "Process Creation (ID 1)" = (Get-WinEvent -FilterHashtable
@{LogName='Microsoft-Windows-Sysmon/Operational'; Id=1} -MaxEvents 1000 -
ErrorAction SilentlyContinue).Count
    "Network Connection (ID 3)" = (Get-WinEvent -FilterHashtable
@{LogName='Microsoft-Windows-Sysmon/Operational'; Id=3} -MaxEvents 1000 -
ErrorAction SilentlyContinue).Count
    "File Creation (ID 11)" = (Get-WinEvent -FilterHashtable
@{LogName='Microsoft-Windows-Sysmon/Operational'; Id=11} -MaxEvents 1000
-ErrorAction SilentlyContinue).Count
    "Registry Events (ID 12-14)" = (Get-WinEvent -FilterHashtable
@{LogName='Microsoft-Windows-Sysmon/Operational'; Id=12,13,14} -MaxEvents
1000 -ErrorAction SilentlyContinue).Count
}

$eventCounts.GetEnumerator() | ForEach-Object {
    Write-Host "$($_.Key): $($_.Value) events" -ForegroundColor Cyan

}
```

Part 2: Log Collection & Parsing

Step 2.1: Export SYSMON Logs

# Create export_logs.ps1:

```
# Set output path
$outputPath = "C:\CTI_Pipeline\logs\sysmon_events.csv"
New-Item -ItemType Directory -Force -Path "C:\CTI_Pipeline\logs"
```

```
# Export all Sysmon events (adjust MaxEvents for production)
Write-Host "Exporting Sysmon events..." -ForegroundColor Yellow
Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational" -MaxEvents
50000 -ErrorAction SilentlyContinue |
    Select-Object TimeCreated, Id, Message,
@{Name='XML';Expression={$_.ToXml()}} |
    Export-Csv -Path $outputPath -NoTypeInformation -Encoding UTF8

Write-Host "Exported to: $outputPath" -ForegroundColor Green

Write-Host "Total events exported: $((Import-Csv $outputPath).Count)" -
ForegroundColor Cyan
```

Command to Run the script:

```
.\export_logs.ps1
```

Step 2.2: Parse XML Structure

`# Create parse_sysmon_logs.py`:

```
import pandas as pd
import xml.etree.ElementTree as ET
import re
from datetime import datetime
import json

def parse_sysmon_xml(xml_string):
    """Parse Sysmon event XML and extract key fields"""
    try:
        root = ET.fromstring(xml_string)

        # Define namespace
        ns = {'event':
'http://schemas.microsoft.com/win/2004/08/events/event'}

        # Extract system data
        system = root.find('.//event:System', ns)
        event_id = system.find('.//event:EventID', ns).text if
system.find('.//event:EventID', ns) is not None else None
        time_created = system.find('.//event:TimeCreated',
ns).get('SystemTime') if system.find('.//event:TimeCreated', ns) is not
None else None

        # Extract event data
        event_data = {}
        data_elements = root.findall('.//event:EventData/event:Data', ns)

        for data in data_elements:
            name = data.get('Name')
            value = data.text if data.text else ''
            event_data[name] = value

        return {
            'event_id': event_id,
            'timestamp': time_created,
            'process_id': event_data.get('ProcessId', ''),
            'command_line': event_data.get('CommandLine', ''),
            'image': event_data.get('Image', ''),
            'parent_image': event_data.get('ParentImage', ''),
            'user': event_data.get('User', ''),
            'network_destination': event_data.get('DestinationIp', '') +
':' + event_data.get('DestinationPort', ''),
            'hash': event_data.get('Hashes', ''),
            'target_filename': event_data.get('TargetFilename', ''),
            'registry_target': event_data.get('TargetObject', ''),
            'raw_data': json.dumps(event_data)
        }
    except Exception as e:
        print(f"Error parsing XML: {e}")
        return None

def process_sysmon_logs(csv_path):
    """Process exported Sysmon CSV and create structured dataset"""
    print("Loading CSV file...")
    df = pd.read_csv(csv_path)

    print(f"Total events loaded: {len(df)}")

    # Parse XML for each event
    parsed_events = []
    for idx, row in df.iterrows():
        if idx % 1000 == 0:
            print(f"Processing event {idx}/{len(df)}...")

        parsed = parse_sysmon_xml(row['XML'])
        if parsed:
            parsed_events.append(parsed)

    # Create structured dataframe
    structured_df = pd.DataFrame(parsed_events)

    # Clean timestamp
    structured_df['timestamp'] =
pd.to_datetime(structured_df['timestamp'])

    # Remove duplicate events
    structured_df = structured_df.drop_duplicates(subset=['timestamp',
'process_id', 'command_line'])

    print(f"\n Parsed {len(structured_df)} unique events")
    print(f"Event
distribution:\n{structured_df['event_id'].value_counts()}")

    return structured_df

if __name__ == "__main__":
    # Process logs
    csv_path = r"C:\CTI_Pipeline\logs\sysmon_events.csv"
    output_path = r"C:\CTI_Pipeline\logs\structured_events.csv"

    df = process_sysmon_logs(csv_path)

    # Save structured dataset
    df.to_csv(output_path, index=False)
    print(f"\n Saved structured dataset to: {output_path}")

    # Display sample
    print("\n=== Sample Events ===")
    print(df[['event_id', 'timestamp', 'image',
'command_line']].head(10))
```

Command to Run the parser:

```
python parse_sysmon_logs.py
```

Part 3: Vectorization Pipeline

Step 3.1: Text Preprocessing

# Create preprocess_text.py:

```
import re
import pandas as pd
from pathlib import Path

def clean_command_line(cmd):
    """Clean and normalize command line text"""
    if not isinstance(cmd, str) or cmd == '':
        return ''

    # Convert to lowercase
    cmd = cmd.lower()

    # Remove paths, keep only executable names
    cmd = re.sub(r'[c-z]:\\[^\s]*\\([^\\]+\.exe)', r'\1', cmd)

    # Remove specific file paths but keep general structure
    cmd = re.sub(r'\\users\\[^\\]+\\', r'\\users\\<user>\\', cmd)
    cmd = re.sub(r'\\temp\\[^\\]+', r'\\temp\\<file>', cmd)

    # Remove UUIDs and GUIDs
    cmd = re.sub(r'\{[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-
9a-f]{12}\}', '<guid>', cmd)

    # Remove long hex strings
    cmd = re.sub(r'\b[0-9a-f]{32,}\b', '<hash>', cmd)

    # Normalize whitespace
    cmd = ' '.join(cmd.split())

    return cmd

def extract_executable_name(path):
    """Extract executable name from full path"""
    if not isinstance(path, str) or path == '':
        return ''

    return Path(path).name.lower()

def preprocess_dataset(df):
    """Preprocess entire dataset"""
    print("Preprocessing text fields...")
    # Clean command lines
    df['command_line_cleaned'] =
df['command_line'].apply(clean_command_line)

    # Extract executable names
    df['image_name'] = df['image'].apply(extract_executable_name)
    df['parent_image_name'] =
df['parent_image'].apply(extract_executable_name)

    # Create combined text for embedding
    df['combined_text'] = (
        df['image_name'] + ' ' +
        df['command_line_cleaned'] + ' ' +
        df['parent_image_name']
    )

    # Remove empty combined text
    df = df[df['combined_text'].str.strip() != '']

    print(f" Preprocessed {len(df)} events")

    return df

if __name__ == "__main__":
    # Load structured events
    df = pd.read_csv(r"C:\CTI_Pipeline\logs\structured_events.csv")

    # Preprocess
    df_processed = preprocess_dataset(df)

    # Save
    df_processed.to_csv(r"C:\CTI_Pipeline\logs\preprocessed_events.csv",
index=False)
    print("\n=== Sample Preprocessed Events ===")
    print(df_processed[['event_id', 'image_name',
'command_line_cleaned']].head())
```

Step 3.2: Generate Embeddings

`# Create generate_embeddings.py`:

```
import pandas as pd
import numpy as np
from sentence_transformers import SentenceTransformer
from sklearn.preprocessing import OneHotEncoder
import pickle
from tqdm import tqdm

def load_embedding_model(model_name='sentence-transformers/all-MiniLM-L6-
v2'):
    """Load pre-trained embedding model"""
    print(f"Loading embedding model: {model_name}")
    model = SentenceTransformer(model_name)
    return model

def generate_text_embeddings(texts, model, batch_size=32):
    """Generate embeddings for text data"""
    print(f"Generating embeddings for {len(texts)} texts...")

    # Convert to list and handle empty strings
    text_list = [str(t) if t else "unknown" for t in texts]

    # Generate embeddings in batches
    embeddings = model.encode(
        text_list,
        batch_size=batch_size,
        show_progress_bar=True,
        convert_to_numpy=True
    )

    return embeddings

def encode_categorical_features(df):
    """One-hot encode categorical features"""
    print("Encoding categorical features...")

    # Encode event_id
    encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
    event_id_encoded = encoder.fit_transform(df[['event_id']])

    # Save encoder for later use
    with open(r"C:\CTI_Pipeline\models\event_id_encoder.pkl", 'wb') as f:
        pickle.dump(encoder, f)

    return event_id_encoded, encoder

def create_combined_vectors(text_embeddings, categorical_features):
    """Combine text embeddings with categorical features"""
    print("Combining feature vectors...")

    combined = np.hstack([text_embeddings, categorical_features])

    print(f" Created combined vectors of shape: {combined.shape}")
    return combined

if __name__ == "__main__":
    import os
    os.makedirs(r"C:\CTI_Pipeline\models", exist_ok=True)
    os.makedirs(r"C:\CTI_Pipeline\embeddings", exist_ok=True)

    # Load preprocessed data
    df = pd.read_csv(r"C:\CTI_Pipeline\logs\preprocessed_events.csv")

    print(f"Loaded {len(df)} events")

    # Load embedding model
    model = load_embedding_model()

    # Generate text embeddings
    text_embeddings = 
generate_text_embeddings(df['combined_text'].tolist(), model)

    # Encode categorical features
    categorical_encoded, encoder = encode_categorical_features(df)

    # Combine vectors
    final_vectors = create_combined_vectors(text_embeddings, 
categorical_encoded)

    # Save embeddings
    np.save(r"C:\CTI_Pipeline\embeddings\event_embeddings.npy",
final_vectors)

    # Save metadata
    df[['event_id', 'timestamp', 'image_name',
'command_line_cleaned']].to_csv(
        r"C:\CTI_Pipeline\embeddings\event_metadata.csv",
        index=False
    )

    print(f"\n Saved embeddings: {final_vectors.shape}")

    print(f" Embedding dimension: {final_vectors.shape[1]}")
```

Command to Run the Embedding Generation:

```
python preprocess_text.py
python generate_embeddings.py
```

Part 4: Vector Database Storage

Step 4.1: Setup ChromaDB

`# Create setup_vectordb.py:`

```
import chromadb
from chromadb.config import Settings
import pandas as pd
import numpy as np
from datetime import datetime
import json

def initialize_chromadb(persist_directory="C:\\CTI_Pipeline\\vectordb"):
    """Initialize ChromaDB client"""
    print("Initializing ChromaDB...")

    client = chromadb.PersistentClient(
        path=persist_directory,
        settings=Settings(
            anonymized_telemetry=False,
            allow_reset=True
        )
    )

    return client

def create_collection(client, collection_name="sysmon_events"):
    """Create or get collection"""
    print(f"Creating collection: {collection_name}")

    # Delete if exists (for fresh start)
    try:
        client.delete_collection(name=collection_name)
    except:
        pass
    collection = client.create_collection(
        name=collection_name,
        metadata={"description": "SYSMON events with embeddings"}
    )

    return collection

def batch_insert_vectors(collection, embeddings, metadata_df,
batch_size=1000):
    """Insert vectors in batches"""
    print(f"Inserting {len(embeddings)} vectors in batches of
{batch_size}...")

    total_batches = (len(embeddings) + batch_size - 1) // batch_size

    for i in range(0, len(embeddings), batch_size):
        batch_end = min(i + batch_size, len(embeddings))
        batch_num = i // batch_size + 1

        print(f"Processing batch {batch_num}/{total_batches}...")

        # Prepare batch data
        batch_embeddings = embeddings[i:batch_end].tolist()
        batch_metadata = metadata_df.iloc[i:batch_end]

        # Create IDs
        ids = [f"event_{i+j}" for j in range(len(batch_embeddings))]

        # Create metadata dicts
        metadatas = []
        for idx, row in batch_metadata.iterrows():
            meta = {
                'event_id': str(row['event_id']),
                'timestamp': str(row['timestamp']),
                'image_name': str(row['image_name']) if
pd.notna(row['image_name']) else '',
                'label': 'unlabeled'  # Will be updated in labeling phase
            }
            metadatas.append(meta)

        # Create documents (text representation for search)
        documents = [
            f"{row['image_name']} {row['command_line_cleaned']}"
            for _, row in batch_metadata.iterrows()
        ]

        # Insert batch
        collection.add(
            ids=ids,
            embeddings=batch_embeddings,
            metadatas=metadatas,
            documents=documents
        )

    print(f" Inserted {len(embeddings)} vectors successfully")

def create_index(collection):
    """ChromaDB automatically indexes, but we can verify"""
    count = collection.count()
    print(f" Collection contains {count} vectors")
    print(f" Index ready for similarity search")
```

```
if __name__ == "__main__":
    # Load embeddings and metadata
    embeddings =
np.load(r"C:\CTI_Pipeline\embeddings\event_embeddings.npy")
    metadata_df =
pd.read_csv(r"C:\CTI_Pipeline\embeddings\event_metadata.csv")

    print(f"Loaded {len(embeddings)} embeddings")

    # Initialize ChromaDB
    client = initialize_chromadb()

    # Create collection
    collection = create_collection(client)

    # Insert vectors
    batch_insert_vectors(collection, embeddings, metadata_df)

    # Verify index
    create_index(collection)

    # Test query
    print("\n=== Testing Similarity Search ===")
    results = collection.query(
        query_embeddings=[embeddings[0].tolist()],
        n_results=5
    )

    print("Top 5 similar events:")
    for i, (doc, meta) in enumerate(zip(results['documents'][0],
results['metadatas'][0])):

        print(f"{i+1}. Event ID: {meta['event_id']}, Image:
{meta['image_name']}")
```

Command to Run the Vector Database Setup:

```
python setup_vectordb.py
```

Part 5: Labeling Strategy

Step 5.1: Generate Benign/Normal Data

`# Create collect_benign_data.ps1:`

```
# Collect benign data during normal operations
Write-Host "Starting benign data collection (30 minutes)..." -
ForegroundColor Yellow
Write-Host "Perform normal system activities: browse web, open
applications, edit documents" -ForegroundColor Cyan

# Capture start time
$startTime = Get-Date
$duration = 30  # minutes
```

```
# Wait for data collection
Start-Sleep -Seconds ($duration * 60)

# Export benign events
$outputPath = "C:\CTI_Pipeline\logs\benign_events.csv"
Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational" -MaxEvents
10000 |
    Where-Object { $_.TimeCreated -gt $startTime } |
    Select-Object TimeCreated, Id, Message,
@{Name='XML';Expression={$_.ToXml()}} |
    Export-Csv -Path $outputPath -NoTypeInformation


Write-Host "Collected benign events" -ForegroundColor Green
```

Step 5.2: Simulate Malicious/Abnormal Activities

`# Create simulate_malicious.ps1:`

```
# This simulates MITRE ATT&CK techniques

Write-Host "Simulating malicious activities..." -ForegroundColor Red

# T1059.003: Command and Scripting Interpreter: Windows Command Shell
cmd.exe /c "whoami && ipconfig && net user"

# T1082: System Information Discovery
systeminfo
wmic computersystem get domain

# T1083: File and Directory Discovery
dir C:\Users /s /b

# T1016: System Network Configuration Discovery
nslookup google.com
netstat -ano

# T1033: System Owner/User Discovery
query user

# T1046: Network Service Scanning (simulated)
Test-NetConnection -ComputerName "127.0.0.1" -Port 445

# Export malicious events
$outputPath = "C:\CTI_Pipeline\logs\malicious_events.csv"
Get-WinEvent -LogName "Microsoft-Windows-Sysmon/Operational" -MaxEvents
1000 |
    Select-Object -First 500 TimeCreated, Id, Message,
@{Name='XML';Expression={$_.ToXml()}} |
    Export-Csv -Path $outputPath -NoTypeInformation


Write-Host "Collected simulated malicious events" -ForegroundColor Green
```

Step 5.3: Label Events with MITRE ATT&CK

# Create label_events.py:

```
import pandas as pd
import chromadb
from chromadb.config import Settings
import json

# MITRE ATT&CK technique mapping
ATTACK_PATTERNS = {
    'T1059.003': {
        'name': 'Command and Scripting Interpreter: Windows Command 
Shell',
        'patterns': ['cmd.exe', 'powershell.exe'],
        'keywords': ['whoami', 'ipconfig', 'net user', 'net group']
    },
    'T1082': {
        'name': 'System Information Discovery',
        'patterns': ['systeminfo', 'wmic'],
        'keywords': ['computersystem', 'os get']
    },
    'T1083': {
        'name': 'File and Directory Discovery',
        'patterns': ['dir', 'tree'],
        'keywords': ['/s', '/b', 'recurse']
    },
    'T1016': {
        'name': 'System Network Configuration Discovery',
        'patterns': ['ipconfig', 'nslookup', 'netstat'],
        'keywords': ['all', 'ano']
    },
    'T1033': {
        'name': 'System Owner/User Discovery',
        'patterns': ['whoami', 'query user', 'quser'],
        'keywords': []
    }
}

def detect_attack_technique(command_line, image_name):
    """Detect MITRE ATT&CK technique based on command"""
    if not isinstance(command_line, str):
        return None, None
    
    command_lower = command_line.lower()
    image_lower = image_name.lower() if isinstance(image_name, str) else 
''
    
    for technique_id, technique_info in ATTACK_PATTERNS.items():
        # Check if image matches known patterns
        for pattern in technique_info['patterns']:
            if pattern in image_lower or pattern in command_lower:
                # Check for specific keywords
                if technique_info['keywords']:
                    if any(keyword in command_lower for keyword in 
technique_info['keywords']):
                        return technique_id, technique_info['name']
                else:
                    return technique_id, technique_info['name']
    return None, None

def label_dataset(benign_csv, malicious_csv, output_csv):
    """Label benign and malicious events"""
    print("Loading datasets...")

    # Load and parse both datasets
    benign_df = pd.read_csv(benign_csv)
    malicious_df = pd.read_csv(malicious_csv)

    # Parse if needed (assuming already parsed)
    # For this example, we'll use the preprocessed data

    # Load preprocessed events
    all_events =
pd.read_csv(r"C:\CTI_Pipeline\logs\preprocessed_events.csv")

    # Label all as benign initially
    all_events['label'] = 'benign'
    all_events['mitre_technique'] = ''
    all_events['mitre_tactic'] = ''

    # Detect and label malicious patterns
    print("Detecting malicious patterns...")
    for idx, row in all_events.iterrows():
        technique_id, technique_name = detect_attack_technique(
            row['command_line_cleaned'],
            row['image_name']
        )

        if technique_id:
            all_events.at[idx, 'label'] = 'malicious'
            all_events.at[idx, 'mitre_technique'] = technique_id
            all_events.at[idx, 'mitre_tactic'] = technique_name

    # Statistics
    label_counts = all_events['label'].value_counts()
    print(f"\nLabeling Summary:")
    print(f"  Benign: {label_counts.get('benign', 0)}")
    print(f"  Malicious: {label_counts.get('malicious', 0)}")

    if 'malicious' in label_counts:
        print(f"\nMITRE ATT&CK Techniques Detected:")
        technique_counts = all_events[all_events['label'] ==
'malicious']['mitre_technique'].value_counts()
        for tech, count in technique_counts.items():
            print(f"  {tech}: {count} events")

    # Save labeled dataset
    all_events.to_csv(output_csv, index=False)
    print(f"\n Saved labeled dataset to: {output_csv}")

    return all_events

def update_vectordb_labels(labeled_df):
    """Update ChromaDB with labels"""
    print("\nUpdating vector database with labels...")

    client = chromadb.PersistentClient(
        path=r"C:\CTI_Pipeline\vectordb",
        settings=Settings(anonymized_telemetry=False)
    )

    collection = client.get_collection("sysmon_events")

    # Update in batches
    batch_size = 1000
    for i in range(0, len(labeled_df), batch_size):
        batch = labeled_df.iloc[i:i+batch_size]

        ids = [f"event_{i+j}" for j in range(len(batch))]

        # Update metadata
        for j, (idx, row) in enumerate(batch.iterrows()):
            try:
                collection.update(
                    ids=[ids[j]],
                    metadatas=[{
                        'event_id': str(row['event_id']),
                        'timestamp': str(row['timestamp']),
                        'image_name': str(row['image_name']) if
pd.notna(row['image_name']) else '',
                        'label': row['label'],
                        'mitre_technique': row['mitre_technique'] if
pd.notna(row['mitre_technique']) else '',
                        'mitre_tactic': row['mitre_tactic'] if
pd.notna(row['mitre_tactic']) else ''
                    }]
                )
            except Exception as e:
                pass  # Skip if event doesn't exist

    print(" Updated vector database with labels")

if __name__ == "__main__":
    # Label events
    labeled_df = label_dataset(
        benign_csv=r"C:\CTI_Pipeline\logs\benign_events.csv",
        malicious_csv=r"C:\CTI_Pipeline\logs\malicious_events.csv",
        output_csv=r"C:\CTI_Pipeline\logs\labeled_events.csv"
    )

    # Update vector database

    update_vectordb_labels(labeled_df)
```