Real-World Data

Public Data (URLs)

- **Source**: URLs are publicly available and can be gathered from various websites, datasets, or traffic data.
- **Model Use**: The model trained on URL data helps detect suspicious or malicious URLs based on features like length, domain type, and keywords, indicating phishing, malware, or other security risks.

Cloud Logs (CloudTrail)

- **Source**: Logs from cloud services (like AWS CloudTrail) provide detailed records of API calls, user activities, and resource usage within the cloud environment.
- **Model Use**: The model trained on CloudTrail logs focuses on identifying misconfigurations, unauthorized access, and unusual activity patterns in the cloud infrastructure, helping improve security posture.

PCAP Files (Network Traffic)

- **Source**: PCAP files are network packet captures from sniffing tools like Wireshark. These files contain raw network traffic.
- **Model Use**: Models trained on network traffic (PCAP files) analyze patterns like unusual data flows, potential intrusions, and unauthorized network access, providing insights into real-time network security.

Each data type (URLs, cloud logs, and network traffic) serves a unique role in security detection, and by using them all, the model can provide a comprehensive view of potential threats across different layers of the system.

URL Data

URL data is widely available, and millions of URLs can be easily collected from various sources. However, the real challenge lies in effectively extracting meaningful features from these URLs for analysis or modeling purposes.

Feature Extraction Process for URL Analysis

Below is a Python function that takes a URL as input and returns a dictionary of the 22 features based on your requirements. These features can be useful for training models or analyzing URL patterns.

```
import pandas as pd
import re
from urllib.parse import urlparse
from tld import get_tld


def process_single_url(url):
    # 1. URL Length
    url_len = len(url)

    # 2. Process domain (TLD extraction)
    def process_tld(url):
        try:
            res = get_tld(url, as_object=True, fail_silently=False, fix_proto
col=True)
            return res.parsed_url.netloc
        except:
            return None

    domain = process_tld(url)

    # 3. Count the number of specific characters in URL
    features = ['@', '?', '-', '=', '.', '#', '%', '+', '$', '!', '*', '"',
',', '//']
    feature_counts = {feature: url.count(feature) for feature in features}

    # 4. Check for abnormal URL pattern (repeating hostname)
    def abnormal_url(url):
        hostname = urlparse(url).hostname
        return 1 if re.search(hostname, url) else 0

    abnormal_url_flag = abnormal_url(url)

    # 5. Check if the URL is using HTTPS
    def httpSecure(url):
        return 1 if urlparse(url).scheme == 'https' else 0

    https_flag = httpSecure(url)

    # 6. Count digits in the URL
    def digit_count(url):
        return sum(1 for char in url if char.isnumeric())

    digit_count_value = digit_count(url)
    # 7. Count letters in the URL
    def letter_count(url):
        return sum(1 for char in url if char.isalpha())

    letter_count_value = letter_count(url)

    # 8. Check if URL is from a shortening service
    def shortening_service(url):
        match = re.search(r'bit\.ly|goo\.gl|t\.co|tinyurl|adf\.ly|url4\.eu|sh
ort\.to|qr\.net|1url\.com', url)
        return 1 if match else 0

    shortening_flag = shortening_service(url)

    # 9. Count the number of directories in the URL path
    def no_of_dir(url):
        urldir = urlparse(url).path
        return urldir.count('/')

    dir_count = no_of_dir(url)

    # 10. Check for suspicious words in URL (e.g., 'login', 'paypal')
    def suspicious_words(url):
        match = re.search(r'PayPal|login|signin|bank|account|update|free|serv
ice|bonus|ebayisapi|webscr', url)
        return 1 if match else 0

    suspicious_flag = suspicious_words(url)

    # 11. Calculate hostname length
    hostname_length = len(urlparse(url).netloc)

    # 12. Count the number of uppercase letters in the URL
    upper_count = sum(1 for char in url if char.isupper())

    # 13. Count the number of lowercase letters in the URL
    lower_count = sum(1 for char in url if char.islower())

    # 14. Check if the URL has a "www" prefix
    has_www = 1 if 'www.' in url else 0

    # 15. Count number of subdomains (split by '.')
    subdomain_count = len(urlparse(url).hostname.split('.')) - 2 if urlparse
(url).hostname else 0
```

```
    # 16. Count the number of query parameters
    query_count = len(urlparse(url).query.split('&')) if urlparse(url).query
else 0

    # 17. Count the number of fragments in the URL
    fragment_count = 1 if urlparse(url).fragment else 0

    # 18. Check if the URL uses a port number
    has_port = 1 if urlparse(url).port else 0

    # 19. Count the number of slashes in the URL
    slash_count = url.count('/')

    # 20. Check if the URL uses a path
    has_path = 1 if urlparse(url).path else 0

    # 21. Check if the URL contains "http"
    contains_http = 1 if 'http' in url else 0

    # 22. Check if the URL contains a valid top-level domain
    valid_tld = 1 if process_tld(url) else 0

    # 23. Check if the URL contains a valid domain (e.g., example.com)
    has_valid_domain = 1 if domain else 0

    # 24. Check if the URL contains the string "secure"
    contains_secure = 1 if 'secure' in url else 0

    # 25. Check if the URL contains the string "login"
    contains_login = 1 if 'login' in url else 0

    # 26. Check if the URL contains the string "signup"
    contains_signup = 1 if 'signup' in url else 0

    # Combine all features into a dictionary
    features_dict = {
        'url_len': url_len,
        '@': feature_counts['@'],
        '?': feature_counts['?'],
        '-': feature_counts['-'],
        '=': feature_counts['='],
        '.': feature_counts['.'],
        '#': feature_counts['#'],
        '%': feature_counts['%'],
        '+': feature_counts['+'],
        '$': feature_counts['$'],
        '!': feature_counts['!'],
        '*': feature_counts['*'],
        ',': feature_counts[','],
        '//': feature_counts['//'],
        'abnormal_url': abnormal_url_flag,
        'https': https_flag,
        'digits': digit_count_value,
        'letters': letter_count_value,
        'Shortening_Service': shortening_flag,
        'count_dir': dir_count,
        'sus_url': suspicious_flag,
        'hostname_length': hostname_length
    }

    # Convert to a DataFrame (for easier handling and saving)
    df_single = pd.DataFrame([features_dict])

    #df_single['Category'] = -1  # Here, we set -1 because it's unknown for a
single URL
    # Save to CSV
    df_single.to_csv('single_url_test.csv', index=False)

    return df_single


# Example usage:
url = "http://example.com/login?user=test"
df_single = process_single_url(url)
print(df_single)
```

Explanation of Features

1. **url_len**: The total length of the URL. Longer URLs may sometimes indicate complex queries or potential obfuscation.
2. **@**: Count of the "@" symbol in the URL. Its presence might indicate the URL is an email address or the use of mail-related services.
3. **?**: Counts the "?" symbol, typically found in URLs with query parameters.
4. : Counts hyphens, which might separate parts of the domain name or path.
5. **=**: Counts the equal sign ("="), often used in URL parameters to assign values to variables.
6. **.**: Counts the number of periods (dots), which are common in domain names and file extensions.

7. **#**: Counts the hash (#) symbol, commonly used for fragment identifiers or anchors within a page.
8. **%**: Counts percent signs, which are used in URL encoding for special characters.
9. **+**: Counts plus signs, often used to represent spaces in URLs.
10. **$**: Counts the dollar sign, which may be associated with financial services or scams.
11. **!**: Exclamation mark count, which may be used to highlight or exaggerate a URL's purpose.
12. : Wildcard character count, although rare in URLs, it may be used in domain names or queries.
13. **,**: Comma count, which can be found in certain data URLs or query parameters.
14. **//**: Double-slash occurrences, which can identify the start of the domain or protocol part of the URL.
15. **abnormal URL**: A flag to check if the URL is an IP address (an abnormal pattern in most cases).
16. **https**: A binary feature indicating whether the URL uses the secure HTTPS protocol (1 for HTTPS, 0 for HTTP).
17. **digits**: Count of numeric digits in the URL, which might indicate the presence of a session ID or other identifiers.
18. **letters**: Count of alphabetic characters in the URL, which might be indicative of human-readable names or patterns.
19. **Shortening Service**: A binary feature indicating if the URL uses a known URL shortening service (1 for shortening, 0 for others).
20. **count Dir**: Counts the number of directory-like components in the URL path (indicated by slashes).
21. **sus_url**: A flag for suspicious keywords commonly used in phishing URLs (e.g., PayPal, login).
22. **hostname length**: The length of the domain name (hostname) in the URL, which may indicate its complexity or legitimacy.

Usage Example:

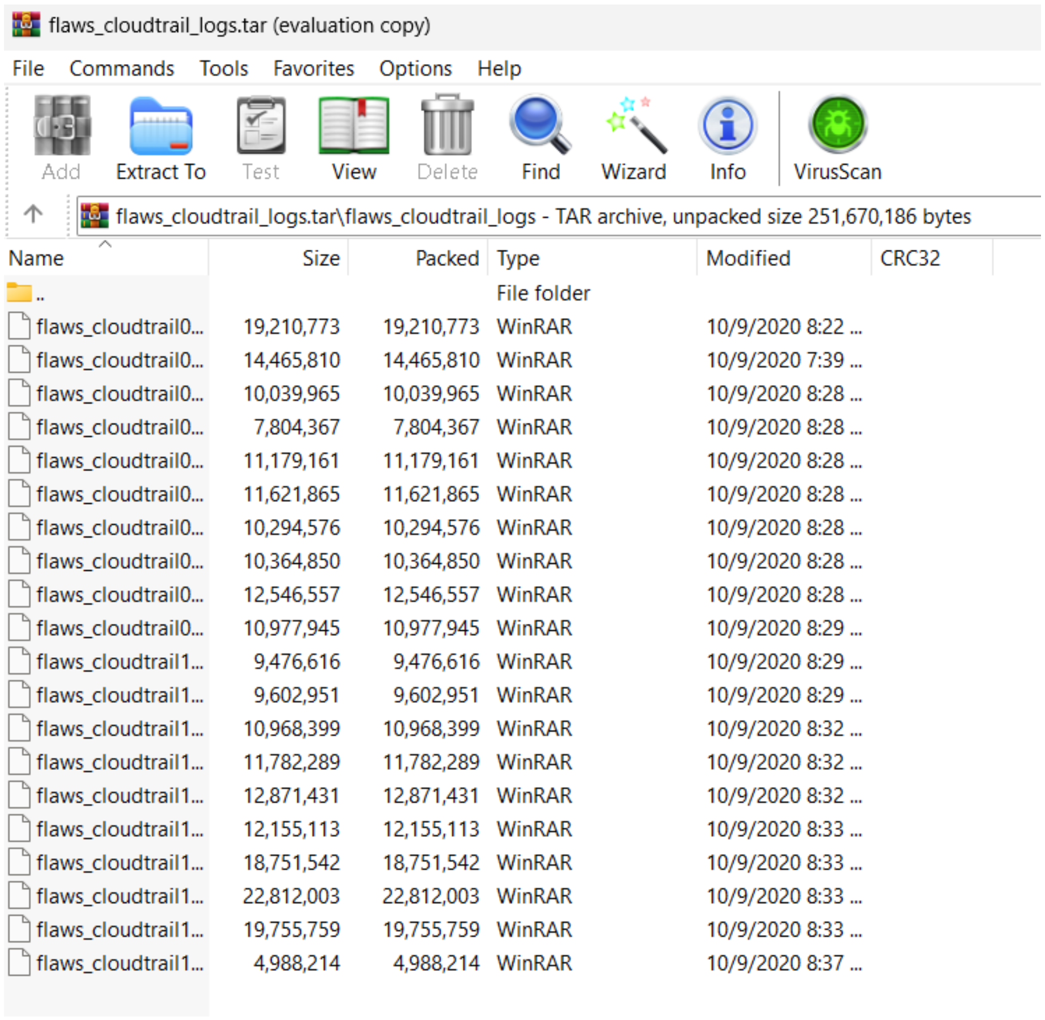
```
python
Copy code
url = "https://www.example.com/login?user=admin&pass=12345"
features = extract_url_features(url)
print(features)
```

This function returns a dictionary with the 22 features, which you can then use for further analysis, training a machine learning model, or other purposes.

Cloud Data

Collecting valid data for training proved to be a real challenge. After considerable research, we discovered that CloudTrail logs provided the most reliable and comprehensive dataset for our needs. In the cloud environment, security issues are primarily caused by misconfigurations rather than traditional network attacks. By training our model using CloudTrail logs, based on best practice configurations, we can effectively detect misconfigurations and enhance security.

AWS CloudTrail is a service that enables governance, compliance, and operational auditing by capturing and recording AWS API calls. It provides detailed logs of user activity, offering deep insights into resource usage and account actions across AWS services. CloudTrail is an essential tool for monitoring and securing AWS environments by logging all actions taken on cloud resources.


flaws_cloudtrail_logs.tar (evaluation copy)
File Commands Tools Favorites Options Help
Add Extract To Test View Delete Find Wizard Info VirusScan

flaws_cloudtrail_logs.tar\flaws_cloudtrail_logs - TAR archive, unpacked size 251,670,186 bytes

Name	Size	Packed	Type	Modified	CRC32
..			File folder		
flaws_cloudtrail0...	19,210,773	19,210,773	WinRAR	10/9/2020 8:22 ...	
flaws_cloudtrail0...	14,465,810	14,465,810	WinRAR	10/9/2020 7:39 ...	
flaws_cloudtrail0...	10,039,965	10,039,965	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	7,804,367	7,804,367	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	11,179,161	11,179,161	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	11,621,865	11,621,865	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,294,576	10,294,576	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,364,850	10,364,850	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	12,546,557	12,546,557	WinRAR	10/9/2020 8:28 ...	
flaws_cloudtrail0...	10,977,945	10,977,945	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	9,476,616	9,476,616	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	9,602,951	9,602,951	WinRAR	10/9/2020 8:29 ...	
flaws_cloudtrail1...	10,968,399	10,968,399	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	11,782,289	11,782,289	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	12,871,431	12,871,431	WinRAR	10/9/2020 8:32 ...	
flaws_cloudtrail1...	12,155,113	12,155,113	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	18,751,542	18,751,542	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	22,812,003	22,812,003	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	19,755,759	19,755,759	WinRAR	10/9/2020 8:33 ...	
flaws_cloudtrail1...	4,988,214	4,988,214	WinRAR	10/9/2020 8:37 ...	

after that we need to change this format for csv file to work in it “and we will do it in testing’

```
import pandas as pd
import json

with open(r'test.json', 'r') as file:
    data = json.load(file)

# If the JSON is a list of records and not a dictionary with a 'Records' key, no
df = pd.json_normalize(data)

# Save the DataFrame to CSV
df.to_csv('output_json.csv', index=False)

# Show the first few rows of the DataFrame
print(df.head())
```

and this's file to test

test.json

Network Data

In network security, we often work with real-world datasets, However, to test a machine learning model with this data, we need to convert **PCAP files into CSV format**. This conversion process involves **extracting relevant features** from the raw network traffic, such as IP addresses, ports, packet length, protocols, and timestamps. These features are then structured in a CSV format, which can be fed into deep learning model for testing and analysis.

and this's code to convert from pcap to csv

```
import pyshark
import csv
import argparse
from collections import defaultdict


def extract_custom_fields(packet, flow_data):
    """Extract custom fields from a packet."""
    try:
        # Extract common fields
        frame_time_delta = float(packet.frame_info.time_delta)  # Time delta in
        flow_data['Flow Duration'] += frame_time_delta * 1e6  # Convert to micro

        # Extract packet direction (forward or backward)
        if hasattr(packet, 'ip') and hasattr(packet, 'tcp'):
            if 'src' in packet.ip.field_names and 'dst' in packet.ip.field_names
                if packet.ip.src < packet.ip.dst:
                    flow_data['Total Fwd Packets'] += 1
                    flow_data['Fwd Packets'] += int(packet.tcp.len or 0)
                else:
                    flow_data['Bwd Packets'] += int(packet.tcp.len or 0)

        # SYN/FIN flags
        if hasattr(packet, 'tcp'):
            flow_data['SYN Flag Count'] += int(packet.tcp.flags_syn == '1')
            flow_data['FIN Flag Count'] += int(packet.tcp.flags_fin == '1')

        # Update timestamps for Active Mean and Idle Mean
        timestamp = float(packet.frame_info.time_epoch)
        flow_data['timestamps'].append(timestamp)

    except AttributeError:
        pass


def finalize_features(flow_data):
    """Calculate derived features like Active Mean and Idle Mean."""
    timestamps = flow_data['timestamps']

    if len(timestamps) > 1:
        time_differences = [timestamps[i] - timestamps[i - 1] for i in range(1,

        # Active Mean: Time differences less than 1 second
        active_times = [t for t in time_differences if t < 1]
        flow_data['Active Mean'] = sum(active_times) / len(active_times) if acti

        # Idle Mean: Time differences greater than or equal to 1 second
        idle_times = [t for t in time_differences if t >= 1]
        flow_data['Idle Mean'] = sum(idle_times) / len(idle_times) if idle_times
    else:
        flow_data['Active Mean'] = 0
        flow_data['Idle Mean'] = 0

    # Remove timestamps (no need to output)
    del flow_data['timestamps']


def pcap_to_csv(input_file, output_file):
    """Convert PCAP file to CSV with custom fields."""
    # Open the PCAP file
    capture = pyshark.FileCapture(input_file)

    headers = [
        'Flow Duration', 'Total Fwd Packets', 'Fwd Packets', 'Bwd Packets',
        'Flow IAT', 'SYN Flag Count', 'FIN Flag Count', 'Active Mean', 'Idle Mea
    ]

    with open(output_file, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=headers)
        writer.writeheader()

        # Flow data storage
        flow_data = defaultdict(lambda: 0)
        flow_data['timestamps'] = []

        for packet_number, packet in enumerate(capture, start=1):
            extract_custom_fields(packet, flow_data)

            if packet_number % 100 == 0:
                print(f"Processed {packet_number} packets")

        # Finalize and write features
        finalize_features(flow_data)
        flow_data['Label'] = 'BENIGN'  # Update this dynamically if needed
        writer.writerow(flow_data)

    print(f"Conversion complete. Output saved to {output_file}")


input_file = "test.pcap"  # Replace with your PCAP file path
output_file = "output_pcap.csv"  # Replace with your desired output CSV path
pcap_to_csv(input_file, output_file)
```

and this’s file to test

test.pcap

The data collection process and answering how we will use and test these models is complete. Now, let's dive deeper into the models.