Intrusion Detection System Project Documentation

Adam Sin, Eliyahu Fridman

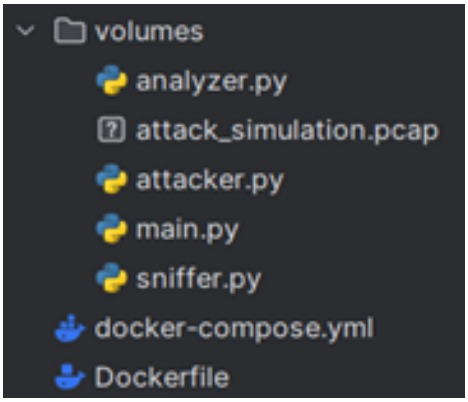
2024

Contents

1 Introduction

Our goal was to build an Intrusion Detection System (IDS) focusing on detecting packets suspicious of data exfiltration. We specifically examined the structure of packets to identify anomalies that could indicate malicious activities. The IDS was designed to sniff traffic on an interface, analyze each packet, and add the packet's general information to a list of sniffed packets, flagging those that failed certain tests along with the reason.

2 Project Structure



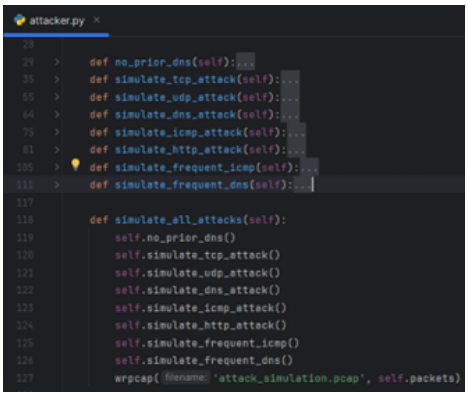
2.1 Docker Setup

The IDS is being run inside a Docker container for several reasons:

- **Isolation**: By isolating the IDS inside a container, we ensure that only the intended traffic is analyzed. When running tests using TCP replay, we avoid interference from unrelated network packets.
- **Docker-Compose and Dockerfile**: The Docker Compose file set up the container, while the Dockerfile specified the necessary dependencies, such as `tkinter`, `pyshark`, and more.

2.2 Attack Simulation (`attacker.py`)

This script's only purpose is to create `attack_simulation.pcap` which is used for the TCP replay testing each and every test the analyzer conducts on the packets. The script creates a list with 132 packets, most of them meant to be suspicious of data exfiltration while a small amount are valid and their purpose will be explained later. The list is them used to create the pcap file using scapy's wrpcap function. All of the packets are outgoing, meaning they are sent from our company's internal IPs to unknown external IPs. Because internal traffic shouldn't cause data exfiltration unless we have a physical mole, external traffic has nothing to do with our company, and the assignment says to not test ingoing packets (from outside in).



2.3 Sniffer (`sniffer.py`)

The Sniffer class, implemented in **`sniffer.py`**, is responsible for sniffing and storing network packets using **`pyshark`**'s **`LiveCapture`** function. The sniffer maintains a queue as a buffer, which the analyzer later uses to test the packets. It also includes a flag that determines whether to continue sniffing or stop.

```
class Sniffer:
    def __init__(self):
        self.sniffing = False
        self.buffer = queue.Queue()

    def sniff_packets(self, interface='eth0'):
        self.sniffing = True
        print(f"Sniffing on interface {interface}...")
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        self.capture = pyshark.LiveCapture(interface=interface)
        try:
            for packet in self.capture.sniff_continuously():
                if not self.sniffing:
                    break
                self.buffer.put(packet)
        finally:
            loop.run_until_complete(self.capture.close_async())
            loop.close()  # add packet to buffer
```

2.4 Analyzer (`analyzer.py`)

The Analyzer class, implemented in **`analyzer.py`**, validates the captured packets by testing each packet to determine if it exhibits suspicious behavior. The **`validate`** function processes packets while the sniffer is active and there are packets to validate.
IsValid function is used to determine whether each packet is valid. It invokes several protocol-specific handlers to perform these checks:

```
class Analyzer:
    def __init__(self):
        self.flows = []
        self.internalIPs = ['127.0.0.1']
        self.icmp_attempts = {}  # Track outgoing connection attempt
        self.dns_attempts = {}   # Track outgoing connection attempt
        self.http_attempts = {}  # Track outgoing connection attempt

    def validate(self, sniffer):...
    def isValid(self, p):...
    def isInternal(self, ip):...

    def tcp_handle(self, p):...
    def udp_handle(self, p):...
    def dns_handle(self, p):...
    def icmp_handle(self, p):...
    def http_handle(self, p):...
    def validate_tls(self, p):...

    def DNS_set(self, dst_ip):...
    def track_attempts(self,dst_ip,attempts,window=10,threshold=10):
```

- **Is an outgoing packet?**:
  packets that are not outgoing don't interest us:

```
if self.isInternal(p.ip.src) and self.isInternal(p.ip.dst):
    return True, "Internal traffic - skipping"
if not self.isInternal(p.ip.src) and not self.isInternal(p.ip.dst):
    return True, "External traffic - skipping"
if not self.isInternal(p.ip.src) and self.isInternal(p.ip.dst):
    return True, "Incoming traffic - skipping"
```

- **TCP Handler**:
  Checks for usual tcp packet ports for secure traffic. Unusual ports may indicate tries to leak data unnoticed. More secure ports can and should be added.

```
if p.tcp.dstport not in ['80','443','53']:  # Only allo
    return False, f"Invalid TCP port: {p.tcp.dstport}"
```

  Tests for HTTPS packets on TCP protocol

```
if p.tcp.dstport == '443':
```

  A https packet must pass security tests, which is its purpose (tls tests are explained later)

```
tlsValid, reason = self.validate_tls(p)
if not tlsValid:  # Combine TLS validat
    return False, reason
```

  https packets sized more than 1460 bytes is unusual, we won't allow sending so much data.

```
elif len(p) > 1460:
    return False, "Large HTTPS packet"
```

  Unusually large TCP size might be a sign to leaked data in the payload.

```
if len(p) > 1500:  # Typical MTU size for Ethernet
    return False, f"Packet size exceeds typical MTU: {len(p)}"
```

  Both flags that represent both opening and closing a connection are very contrary and might be due to a try to use the server in an unusual way.

```
if (flags & 0x02) and (flags & 0x04):
    return False, "Invalid SYN+RST combination"
```

  PSH flag means packets that should be processed by the receiver before any other given packets in the buffer. A packet too big of that importance might indicate a try to process and get the data fast.

```
if flags & 0x08:
    if len(p) > 1460:
        return False, "Large TCP segment with PUSH flag"
```

  URG flag means packets with some important info that should be processed by the receiver before any other given packets in the buffer. A packet too big of that importance might indicate a try to process and get the data fast.

```
if flags & 0x20:
    if len(p) > 1460:
        return False, "Large TCP segment with URG flag"
```

- **UDP Handler**:
  Checks for usual udp packet ports for secure traffic. Unusual ports may indicate tries to leak data unnoticed. More secure ports can and should be added.

```
if p.udp.dstport not in ['53','123','68']:  # Only allo
    return False, f"Invalid UDP port: {p.udp.dstport}"
```

  NTP packets are for syncing servers and clients' computers. This packet is usually sized 48 bytes so any other number of bytes is suspicious and should be checked for having different data.

```
if p.udp.dstport == '123' and len(p) != 48:
    return False, "Invalid NTP packet"  # NT
```

- **DNS Handler**:
  If the packet is used as DNS

```
if dns_layer is not None:
```

  Any other port than 53, the usual port for DNS might be used for malicious causes. An example is trying to get a DNS response without being detected as a DNS request.

```
if dstport != '53':
    return False, "Invalid DNS response port"
```

  DNS packets with a big payload indicates added data within getting leaked.

```
if len(p[transport_layer].payload) > 512:
    return False, "Possible DNS tunneling"
```

  `track_attempts()` checks how many DNS responses were asked from the same IP and if too many packets in a small window of time might be a try to leak data undetected using many small packets

```
if not self.track_attempts(p["IP"].dst, self.dns_attempts, window=15, threshold=15):
    return False, "High frequency of DNS requests"
```

  if the packet is not a DNS packets with port 53 it may indicate a try to send data using a trusted port to not be detected.

```
elif dstport == '53':
    return False, "Invalid port for none DNS"
```

  For UDP packets that aren't DNS we check that the payload doesn't contain a DNS response. Secretly transmitted DNS responses should alert us.

```
elif hasattr(p,'udp') and hasattr(p.udp,'payload'):
    payload_bytes = p.udp.payload.binary_value
    payload_str = payload_bytes.decode('utf-8', errors='ignore')
    if("com" in payload_str):
        return False, "None DNS port, but DNS response!"
```

To get here means the packet has nothing to do with DNS. If so, `DNS_set()` check that there was a valid DNS response before sending this packet. A case where a packet knew to be sent to our server without a DNS means someone knew in advance our information or were somehow under our radar. Both cases are suspicious and might be intended for data leaks.


```
elif hasattr(p,'udp') and hasattr(p.udp,'payload'):
    payload_bytes = p.udp.payload.binary_value
    payload_str = payload_bytes.decode('utf-8', errors='ignore')
    if("com" in payload_str):
        return False, "None DNS port, but DNS response!"
```


- **ICMP Handler**:
  ICMP packets with a big payload indicates added data within getting leaked.

```
if len(p) > 64:  # Too big for just p
    return False, "Large ICMP packet"
```

`track_attempts()` checks how many ICMP responses were asked from the same IP and if too many packets in a small window of time might be a try to leak data undetected using many small packets

```
if not self.track_attempts(p['IP'].dst, self.icmp_attempts, window=10, threshold=10):
    return False, "High frequency of ICMP requests"
```

- **HTTP Handler**:
  HTTP packets with a big payload indicates added data within getting leaked.

```
if hasattr(p['HTTP'],'file_data') and len(p['HTTP'].file_data) > 900:
    return False, "Large HTTP payload"
```

Host is a mandatory for HTTP 1.1 or higher in order to determine the specific site asking for response. Missing host in the header is unusual and may be to stay undetected.

```
if not hasattr(p['HTTP'],'host'):
    return False, "Missing Host header"
```

HTTP packets with a big header indicates added settings or data within getting leaked.

```
if len(p) > 1300:
    return False, "Unusually large HTTP headers"
```

HTTP packets with these headers mean they contain sensitive information like passwords, permissions, etc. The server should be alerted if it wasn't meant to be sent.

```
for f in ['authorization','cookie','set_cookie','proxy_authorization']:
    if hasattr(p['HTTP'], f):
        return False, f"Sensitive header: {f}"
```

Requests that were sent using potentially malicious scripts may try extract data from the server.

```
if hasattr(p['HTTP'], 'user_agent') and 'python-requests' in p['HTTP'].user_agent.lower():
    return False, f"Suspicious User-Agent: {p['HTTP'].user_agent}"
```

Allows sending data in chunks instead of all at once. Can be used to hide the amount of leaked data.

```
if hasattr(p['HTTP'], 'transfer_encoding') and p['HTTP'].transfer_encoding == 'chunked':
    return False, "Suspicious chunked Transfer-Encoding detected"
```

if the referrer that sent the request isn't a trusted domain, we shouldn't allow him to receive the response.

```
if hasattr(p['HTTP'], 'referer'):
    referer = p['HTTP'].referer
    if not referer.startswith('https://trusted-domain.com'):
        return False, f"Untrusted Referer: {referer}"
```

`track_attempts()` checks how many HTTP responses were asked from the same IP and if too many packets in a small window of time might be a try to leak data undetected using many small packets

```
if not self.track_attempts(p['IP'].dst, self.http_attempts, window=60, threshold=100):
    return False, "High frequency of HTTP requests"
```

- **TLS Handler**:
  A HTTPS packet as to have a TLS layer by definition.

```
if 'TLS' not in p:
    return False, "none/invalid TLS"
```

  Handshake types other than “Client Hello”, “Server Hello” in a TLS layer may be of a malicious intent.

```
if handshake_type not in ['1','2']:
    return False, "none/invalid TLS"
```

  A TLS without a SNI that supposed to contain information about the receiver may be due to security breaches or an untrusted connection.

```
if handshake_type == '1' and not sni:
    return False, f"hostname {getattr(p['TLS'],'certificate_hostname',None)} != SNI {sni}"
```

  if the TLS version is not secure enough might be to try and conduct malicious activity.

```
if version not in ['0x0303', '0x0304']:  # TLS 1.2
    return False, f"invalid TLS version {version}"
```

  if the cipher suite, which is the encryption algorithm aren't secure enough, it may be so it will be easier to read leaked data.

```
if handshake_type == '1' and cipher_suite in ['0x0000', '0x0005']:
    return False, f"Weak cipher suite - {cipher_suite}"
```

2.5 Main Interface (`main.py`)

The main interface, implemented using `tkinter`, integrates the Sniffer, Analyzer, and Attacker components into a cohesive UI.

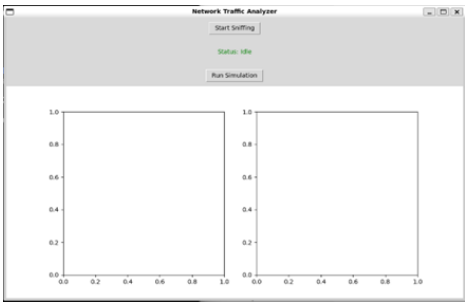
```
class App:
    def __init__(self, root):...

    def mode(self):...
    def start_sniffing(self):...
    def stop_sniffing(self):...

    def run_simulation(self):...

    def update_graph(self):...
    def show_analysis_graph(self, real_time=False):...
    def show_cumulative_packet_graph(self, real_time=False):...

root = tk.Tk()
app = App(root)
root.mainloop()
```

To run the program:

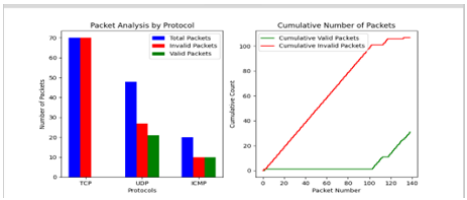
- Download Xlaunch from `https://sourceforge.net/projects/vcxsrv/`
- Run WSL
- Go to the docker-compose.yml and volumes directory.
- docker-compose build
- docker-compose up
- docker exec -it `protocols_pyshark_sniffer_1` bash
- cd volumes
- python3 main.py

Running the program provides the following functionality:

- **Start Sniffing**: will run the sniffer across the interface inside the docker. Another click will stop.
- **Run Simulation**: will use TCP replay to simulate the pcap across the interface so the sniffer could sniff the packets and have the analyzer test them.



- **Statistics Display**: Graphically presents analysis results using Matplotlib, showing cumulative packet counts and protocol breakdowns.



Packets Analysis by protocol:
for each protocol TCP,UDP,ICMP we count:
Blue - the number of packets sniffed
green - the number of valid packets sniffed
red - the number of packets that are suspicious of data exfiltration that were sniffed

Cumulative Number of Packets:
We want to see the cumulative number of valid/invalid packets over time:
Red – the cumulative number of packets that are suspicious of data exfiltration
Green - the cumulative number of valid packets