Intrusion Detection System

Eylon Yaakov Katan[1] **and Noam Leshem**[2]

[1]Ariel University
[2]Ariel University

In this project, we've built an Intrusion Detection System-based application, designed to detect data leak attacks from within an organization. To parse each flow effectively and efficiently, we've parsed each packet to a 5-Tuple: source IP, destination IP, source port, destination port, and the network protocol used. On top of that, each flow was marked in-going or out-going, to help processing later on. The system is designed to return various data about the flow: The amount of data transferred between participants, the protocol used in conversation, and if any attacks were found.

Intrusion Detection System | Attack | Flow | Data Leak
Correspondence: ***eylony1403@gmail.com lesh.noam@gmail.com***

Architecture

The system's logic is based on the Intrusion Detection System products available today. It will listen continuously to the organization's inner network and will try to detect whether a data leak attack occurred, i.e. someone is trying to leak data from within the organization outside.
The system can only alert when it finds an attack, and cannot prevent it. The detection is based upon policies - rules that we have written to decide whether a certain packet sniffed is suspicious. Once a suspicious packet is found, the user will be alerted and will be given all the relevant information regarding the attack.
The system can detect abnormalities in two different ways: First, by opening a pcap/pcapng file, and reading its entire content, while detecting errors. Second, the system allows choosing a Network Interface from all interfaces available on the computer, and sniffing on in, while detecting attacks in real-time.
The graphical user interface was written in Python using a library called flet, this python library was chosen by us because it provides performance and reliability to the back-end of the application, while also having an appealing front-end design for the comfort of the user (see also: https://material.io).
We've set up our docker environment to emulate an isolated network with only one machine running. We can simulate attacks and various kinds of data leak methods without affecting any real and vulnerable machines or computers. We've chosen to use the Hands-On-security Ubuntu-image and have downloaded all the requirements for running flet and scapy. Using a docker-compose yaml file we have set up the simulated network that hosts the docker machine.

Design

Our initial design was divided into a few sections: First, we've implemented a basic method to sniff packets and process each into a 5-tuple. The sniffing was performed using an asynchronous sniffer action, to not interfere with the processing of the packets and the detection process. Later on, we implemented a method to measure the size of data transferred between two sides in each flow, and other metadata to display to the user. All the data was printed to the console to examine whether our actions were correct.
Second, we've implemented the GUI. The graphical user interface has two methods of handling packets: the first one is opening a pcap file and processing the packets within it, and the second one is sniffing on a user-selected interface and processing the packets that flow through that interface in real-time. In both methods, the GUI can filter by each option in the 5-tuple (i.e., filter by source IP, destination IP, source port, destination port, protocol), and the direct of the flow (in-going or out-going). It will update the list of relevant packets accordingly.

Identification Methods

While creating the platform itself, we thought about and researched different types of data leak methods to create a 'rule-book', that can determine whether a packet or a flow is suspicious and alert the user about the danger.
When we sniffed data from a network interface, we decided to examine only outgoing packets, because we are searching for data leaks from within an organization towards the outside world. When we're examining a pcap file, we've decided to check every packet for suspicious behavior, so as not to miss any potential harm.
After research, we decided to test for a few anomalies:
Suspiciously large packets: big packets can be used to hide data within their header/payload. After performing and measuring the data size of various actions (watching YouTube videos, playing online games, etc.) we concluded that a suspiciously large packet would be the size of at least 6000 bytes.
Unmatching Port and Protocol: a packet of a certain protocol that isn't sent to the unique port of the same protocol can indicate that it might be fraudulent, and try to seem like a standard packet while it is not.
BlackLists: It's important to create blacklists for ports, IPs, and protocols to make sure that there are no anomalies regarding them. A network protocol will be blacklisted if it's a local LAN protocol, used to communicate with someone outside of our organization. We chose ARP, LLMNR, and NBNS as an example of such protocols. An IP address will be blacklisted based on knowledge from various sources, such as VirusTotal, AlienVault, and WHOIS. An address will be

needed to insert manually, due to the nature of the program. A port will be blacklisted in the same manner as an IP address.

DNS Tunneling: DNS Tunneling is a data exfiltration method, commonly used by hackers to derive data from an organization. While making a malware attack, before the execution and even to the initial access, the hacker will register a domain and set it to an NS server he owns. During the attack, the malware will gather important data from within the organization (usernames and passwords, files, etc.), encode it (usually using base64), and send a DNS request to the authoritative NS (hacker), embedding the encoded data as a subdomain of the server. That way, the attacker will get the data, and can even send requests back to the malware. Due to the nature of the attack, those requests are sent quite often, and the subdomains used in those are usually large, because of the usage of base64 encoding. To detect DNS Tunneling, we've decided to measure the amount of times a NS server is requested, and the length of the subdomain that the request contains. After searching the web for DNS Tunneling-Using malwares and common password lengths, we concluded that a 15 DNS request that contains subdomains of size 30 is enough to raise suspicion.

ICMP Tunneling: ICMP Tunneling is a technique used by attackers to secretly transmit malicious traffic through network defenses by hiding it within normal-looking ICMP packets. Similar to DNS Tunneling, it exploits the trust placed in ICMP for legitimate diagnostic purposes to bypass security measures. While both techniques leverage trusted protocols, ICMP Tunneling can encapsulate a wider variety of IP traffic, including HTTP, TCP, and SSH packets, making it particularly potent for establishing persistent backdoors and exfiltrating data. Detection requires careful analysis of network traffic patterns and payload contents, as normal-sized ICMP packets may contain hidden malicious data. Awareness of these techniques is crucial for effective cybersecurity strategies. In the same manner, as DNS Tunneling, we're able to detect ICMP Tunneling by searching for an IP that receives many ICMP packets, and all their payload size is bigger than 64 bytes, the standard size.