CHAPTER – IV

Workflow of the Model

4.1 Application Intercepting Agent (AIA):

Application Intercepting agent (AIA) is deployed on each host in the network. It consists of two main modules: Network Interceptor (NI) [61] and Event Logger [72]. The following diagram, Figure 4.1 shows the conceptual framework for NI and event logger:

FIGURE 4.1 Conceptual framework for NI, Event Logger [61]
4.1.1 Network Interceptor (NI):

Network Interceptor [61] performs hooks on system functions, and because of this it captures real-time traffic. Due to this, it detects an attack in real-time.

Network Interceptor [61] has two options on Windows operating system: NDIS Framework, capable of intercepting raw network data and operates at Kernel Level. Another approach works at an intermediate level using WINSOCK Hooking, to intercept socket function calls and socket data. We choose WINSOCK hooking approach, since it provides information of application which has initiated connection, with interception of socket call. Various techniques such as replacing WINSOCK DLL, Hijacking DLL, writing Layered service provider (LSP) driver etc. are available to implement WINSOCK Hooking. Since other techniques leads to DLL version compatibility issues, Microsoft recommends LSP driver implementation [83]. Therefore, we are using WINSOCK 2 SPI Framework to implement Transport Layered Protocol.
We implement NI as a standard DLL. NI is a functional extension to an already existing transport service provider (TSP). We implement NI as a protocol stack of services, in order to provide functions for connection set up, data transfer, to apply flow control, error control, etc. NI is registered with WSCInstallProvider [83], which is an API of SPI framework. In this way, NI can use LSP SPI Architecture of Windows. Once we register this NI, all transport SPI functions implemented by NI are made accessible to ws2_32.dll by callback function mechanism via the LSP's dispatch table. Therefore, NI overrides and intercepts Winsock 2 functions before they are processed by ws2_32.dll. We are mainly interested in functions used by any application to send or receive data. Such
functions are normally send, sendto ,recv, recvfrom, etc. These functions help in intercepting real time data. When any application initiates a connection for the first time, our NI DLL also gets initialized along with it. During the time of initialization, pointers of our callback functions get registered into LSP dispatch table. Therefore, when the application calls any socket functions to initiate a connection or to send/receive data, it first calls our function. Our implemented function obtains the access of socket handle and data which is transmitted. It sends this data to IDPS for inspection and waits for its result. The result is communicated to pattern matching unit via event logger.

4.1.2 Event Logger:

The Event Logger [72] stores the alert log in alert store database. On a high-performance network, the inflow and outflow of events is very high. The speed at which it stores the alert log generally does not match with the flow of incoming and outgoing network packets. In order to store each alert log and not miss any of them, we have implemented an asynchronous architecture. We use the concept of classical producer-consumer scenario. Event Logger is divided into the three functional sub-systems: Event Receiver, Queue and Database Plug-in [72]. Figure 4.3 shows the working of Event Logger:

![Event Logger Diagram](image)

**FIGURE 4.3 Event Logger**
AIA uses UDP protocol to send event log to the Event Logger. Event Receiver receives this log and inserts it in the queue. The database plug-in fetches this log from the queue and stores it in the database.

Figure 4.4 describes the communication flow between PMU and AIA. As shown, a workstation has Process-A running on it. On the same workstation, AIA is deployed. Process-A tries to send a request to a website hosted on the Internet. AIA has a dedicated TCP connection with PMU. Before the packet reaches IDS engine deployed in our network, AIA intercepts the system call at socket level for this connection request. From the socket call intercepted, it obtains all the information about application as well as connection. AIA now sends this information to PMU. PMU verifies this against pertinent application data store and sends the decision of allow/block and AIA allows or drops the connection.

**FIGURE 4.4 Communication flow for AIA**
4.2 Pattern Matching Unit (PMU):

As shown in figure 4.5 and 4.6, PMU has two different deployment modes: inline and out of band. A UDP packet listener will listen on a dedicated UDP port, to gather application and connection information sent by AIA instances. Once a UDP packet is received, listener will store application information and five tuple hash (Source IP, Destination IP, Source Port, Destination Port, Protocol) in a global memory store. Global application store is implemented as a two-dimensional linked list, where first dimension represents a unique system and each system node points to another linked list representing application and connection information for that system.

![FIGURE 4.5 PMU working in inline mode](image)

As stated earlier, when PMU is deployed in out of band mode, it starts TCP server listening on a dedicated port. AIA running on each system initiates a dedicated TCP connection.
from each one of them. All the packets transferred on that system is sent to PMU using this dedicated TCP connection. So, TCP listener will have multiple incoming packet streams. Each packet stream is uniquely identified and differentiated by system identifier. PMU puts a wrapper on every packet to associate it with respective system. It also accumulates all these packets in a common packet queue.

Another process running under TCP listener keeps a watch on this queue. As soon as packet arrives, it sends that packet to IDS engine using standard available interface and API provided by IDS engine. IDS engine inspects the packet for known vulnerabilities using standard signature rule files. If any vulnerability is found, it responds back to the calling application with signature rule id information.

PMU has another data store which provides pertinence of signature to different application or processes. PMU looks up the triggered signature rule id in this data store to fetch the information about pertinent application.
CHAPTER IV Workflow of the Model

From a packet wrapper (Hashvalue of combination of MAC ID & Source IP Address), it retrieves system identifier to lookup second level linked list representing that particular system’s application and associated connection information.

From a packet, it fetches five-tuple information to lookup into retrieved second-level linked list. This lookup provides associated application information. PME now matches associated application information and pertinent application information. If associated application information belongs to this list, then it concludes that application is either compromised or attacked. In that case, PMU stores this information into an Alert store database.

If it doesn’t match, it is considered as a false alarm. Administrator takes a decision on every entry in alert database; mark it as a quarantine decision.

4.3 Aggregation and Correlation Module (ACM):

Host based intrusion detection and prevention systems (IDPS) reacts to different events generated by operating systems running on machines which host these systems. Operating system generates logs of seemingly malicious events for example, when CPU utilization goes high, an application tries to write in system file area or a service is not responding etc. HIDPS generates an alert when such event is reported. An alert generated by HIDPS contains application level information such as hostname, service name, event timestamp, and signature id and signature name [20]. It is because these systems work only at application level and have visibility of only application headers. Network based IDPS, on the contrary, works on network packet headers. In an NIDPS, an IPS sensor is placed at network ingress point, or some other place where one can monitor the packets flowing through entire network. The IDPS sensor monitors network traffic and inspects packet transmissions for malicious behaviour. It generates alert when it observes such malicious behaviour. The alerts generated by NIDPS or perimeter based IDPS contains network level information such as source IP, destination IP, event timestamp, signature id, signature name and protocol details [20]. It is because these systems work at network layer and have visibility of only network protocols and network header details. Such IPS sensors can be placed at various locations along the boundary or periphery of the network. It is then known as Perimeter based IDPS [21]. Perimeter based IDPS is typically deployed at the
entry points of different subnets. The purpose of such deployment is to protect every subnet from generating threats within the network and across the network but within the organization.

This information contained in an alert is crucial for the network administrator to identify the attacker and to take correct preventive decisions for that attack, which includes applying operating system fixes, applying application fixes, blocking of unused network services and so on. However, a single alert may not indicate complex attack scenarios such as DDOS, application targeted attacks followed by port scan, botnets etc. [20] In addition, the number of network packets and log entries that are inspected periodically has a direct impact on number of false alerts generated by an IDS [1].

An alert is an indication given by an IDPS for possibility of an attack. This alert may singularly describe an attack in entirety. However, certain attacks may be distributed across multiple alerts. In our approach, we have considered both the cases. Thus, we aggregate alerts only when it is required.

There are certain alerts, which alone are adequate to provide attack information. An example of such alert is when an insider is aware about certain system vulnerability such as buffer overflow and tries to attack. Our proposed system generates an alert in this case in its entirety. However, for the same vulnerability, it is also possible that an outsider performs port scan followed by application probing to learn about this vulnerability and similar alert is generated. The major difference here is the first case describes the entire attack while second one does not. In the second case, attack instances for port scan, application probing and buffer overflow need to be correlated for complete attack description. The strength of our approach is that it can correlate such events to exactly pinpoint the attack.

**Addressing IDPS issues by Correlating Raw alert log:** Alerts generated by IDPS provide very basic information of attacks, from which it is very difficult to retrieve bigger picture. In order to address complex attack scenarios and generate aggregated alerts related to them, specific issues such as alert flooding, alert context, false alerts and scalability should be addressed. These issues are described as:

- **Alert Flooding:** Alert flooding is caused by multiple alerts generated by IDPS for the same attack instance. For example, when an attacker launches a large ICMP echo attack, it
would be normally continuous attack on a destined system. Alert will be generated for every large ICMP echo packet. The fact that human security expert then studies these alerts intensifies the probability of wrong or inappropriate decision taken. We handle it by collapsing multiple similar alerts into one.

**Alert context:** Context means alerts which are manifested as part of a larger attack sequence. It means relating similar attack instances. The existing detection systems are usually capable of detecting various types of attack, but in between the occurrences of attacks, lay various instances of similar attacks, which the detection systems are not able to differentiate from one another [62]. Our approach handles that as well.

**False alerts:** False positive means alert generated by an IDPS to protect or report system vulnerability which does not exist. Network administrator performs routine maintenance activities by port scan or similar probes. IDPS misclassifies this activity as an alert. Such alerts should be filtered from alert log.

**Scalability:** This issue bothers an IDPS when large number of alerts is generated as network grows and consequently number of probes increases. With proper aggregation, this problem is also handled to an extent by our approach.

Our approach tries to address these issues. We focus primarily on generating only one alert per attack, even if the attack manifested itself into multiple alerts. Such temporal correlation of alerts will reduce the log size considerably. For example, in case of buffer overflow attack, we can group similar alerts on the basis of source IP, destination IP and time interval. For the same example, if external user had tried different attacks for which system is not vulnerable, and IDPS has generated alerts for the same, and then correlation will help to mask them as false alarms. The task of alert information retrieval is handled locally and is sent to a centralized module for further processing. Thus, we successfully manage the trade-off between information gathering and network performance to a large extent.

ACM module covers three aspects of any attack:

1) Attack information – Attack information is basic raw alert log data being sent by any standard IDS. It includes information about attack, severity of attack, network details and signature rule ID which has detected the attack.
2) Application participation in attack – we identify details of attacking and victim applications details like application name, application version, etc.

3) Pertinent application to specific attack – An attack is generally targeted towards some particular application vulnerability. E.g. Attacks for different application servers like Tomcat and Apache will be different even though application protocols are HTTP. Our solution uses signature rule set database with indication of application pertinent to attack.

Figure 4.7 describes our alert aggregation and correlation module (ACM) which is a four stage process: Alert Filtering, Alert Aggregation, Alert masking and root-cause analysis. This module works on Alert Store generated by PMU. This store contains raw alerts with pertinent application information with alert details. This application information is primarily used by our ACM module to correlate alerts into a single alert scenario with precise information about the application name and version responsible for the attack.
Alert Filtering:

There are times when system goes into maintenance mode and requires debugging and trouble shooting. When alert is generated due to activities performed for such trouble shooting applications, it should be filtered because they are irrelevant for attack identification. Administrator has a predefined list of such applications in our system by providing application name and application versions. Any alert generated by this application will be marked for filtering by our system.

Alert Aggregation:

Alert aggregation is merging of events, which are part of common attack, or duplicates in nature. When sequence of attack is generated by same application running on one host or different hosts, it should be aggregated as they are forming a common attack. E.g. If one botnet application is spread across the network and multiple hosts are trying to generate DOS attack to one particular server on internet or multiple servers on internet as a whole, we will aggregate them by application name, application version, variety of destination IPs and attack and this reconstruction will give us attack information of Distributed DOS.

Alert Masking:

When an attacker launches an attack on the nonexistent system or vulnerability, it normally generates false alarms. E.g. If attacker probes that port 80 is open and launches an attack for IIS which is not running on that system. It will generate a false alert by IDPS. When any alert is being generated by our IDPS, it contains information of application which participated in that attack along with applicability of that attack. We compare both the information and mark the false positives being generated.

Root-cause analysis:

This module decides confidence level of each log entry in alert store database to perform correlation of individual alert incidents into meaningful alert scenario. It scans each database entry, which has an initial confidence level set to 1. If an entry has been marked as false positive, this module decreases the confidence level for that entry to 0. Any subsequent entry for same application name, version or same host will result into increase in its confidence level. All such entries will be correlated to form a single threat scenario,
with threat level defined. The final outcome of this step is Incident Report Database with security threat level defined as low, medium, high, critical & fatal.

4.4 Application Quarantine Module (AQM):

Application Quarantine Module uses incident report database for preventive advices. For example, root cause analysis concludes that Mozilla running on machine 20.0.1.15 is being compromised. Administrator marks this application on that host as a quarantine application. Thus, when Application intercepting agent (AIA) asks for permission of network admission or access for mozilla, application quarantine module will look up the alert store to check if mozilla running on 20.0.1.15 is marked as quarantine or not. If match is found then it will reply with BLOCK status and AIA will block the network access for mozilla. So it doesn’t block the access of host machine but only blocks vulnerable application for network access. This improves the network security without compromising network productivity.

4.5 Complete working of all modules:

We now discuss stepwise working of all the modules. We also discuss how they interact with each other in order to achieve our objectives.

Application intercepting agent (AIA) intercepts network traffic being sent or received by any application on the network under consideration. When any application tries to communicate, it opens a socket to send or receive the data. AIA intercepts this traffic and collect details of that application. This application information is passed to pattern matching unit (PMU) along with packet payload of connection.

PMU sends the received network data to intrusion detection engine to scan and detect the attack. Intrusion detection engine can be any general IDPS such as Snort or Suricata [84] [85]. It receives the network information from PMU and scans that traffic for attack using its rule set database. If any attack is found that it generates the alert log and sends it back to PMU. This alert log contains id of signature which matched against the network packet data. Once PMU receives the alert information from intrusion detection engine, it performs lookup on Pertinent Application Information Store database for the signature id of alert. If
a match is found, it verifies the pertinent application for the matched signature id with application associated with the alert.

The application which initiated this malicious connection is called an associated application. The global application store plays an important role of verifying the pertinent application as well as associated application for the matched signature. This is implemented as a two level linked list, snapshot of which is shown in the next section. We have chosen Linked list as a data structure because no of hosts and no of applications running on them can grow or shrink dynamically. In the first level linked list, each node stores a hash value to identify unique host IP Address and host name. Each node of the first linked list points to a set of nodes of second level linked list. Each node of the second level linked list stores connection as well as application information about each connection initiated from that host. PMU performs a lookup in this store. If a match is found, it inserts alert details along with pertinent application information indicated by rule set database in another database called alert store.

Alert aggregation and Correlation module (ACM) works on this alert store database which contains application information and whether the application is pertinent to the alert as described. We have described this module in detail in following sections. ACM generates the final aggregated log which is useful to administrator to take preventive actions and also useful for application quarantine. This unit reduces raw alerts log size by more than 95% and also removes false positives.

Application quarantine module (AQM) works on aggregated alert log. When AQM receives the permission request from an application for network admission or access, it checks for quarantined application information in final aggregated incident report database. If any application is already marked for quarantine then AQM blocks that application’s connection process.

4.6 Workflow Algorithm:

We have discussed each module in detail in previous section. We have discussed how each module communicates with each other at different stages. We now present a complete algorithm of our workflow:

Step 1: AIA intercepts connection request by an application
Step 2: AIA sends connection and application details to PMU along with packet data transferred over that connection.

Step 3: PMU sends connection packet data to IDS engine.

Step 4: If IDS engine scans the data and if vulnerability or attack is found then sends an alert back to PMU. PMU compares associated application of that connection with pertinent application stored in signature database.

Step 5: If successful match, alert details are stored in alert store with confidence level 1.

Step 6: ACM parses each entry in raw alert store and performs the following actions:

   a) Filter alerts generated for network maintenance and trouble-shooting

   b) Mask false positives. In our approach, alerts generated for non-pertinent applications are false positives.

   c) Group alert incidents for profiled applications as a single alert scenario.

   d) Root-cause analysis: This step decides confidence level of each log entry to perform correlation of individual alert incidents into meaningful scenario.

       d.1: If entry is marked as false positive, decrease confidence level to 0.

       d.2: If otherwise, we mark it for further tracking.

       d.3: Whenever next incidence occurs for same application or same host, we increase confidence level and mark it as security threat with threat level.

       d.4: All subsequent log entries from that host for the particular application will be correlated to this incident to form a single threat scenario.

       d.5: The final outcome of this step is Incident Report Database, used by AQM with security threat level defined as low, medium, high, critical & fatal.