Implementing an Object Oriented, Knowledge Based Network Reference Monitor and Intrusion Detection System

*Dr. Alec Yasinsac Yasinsac@cs.fsu.edu
Edwin A. Melendez Melendez@cs.fsu.edu
Sachin Goregaoker Goregaoker@cs.fsu.edu
850.644.6407 (voice)
850.644.0058 (fax)
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530

Abstract

With the unprecedented growth of computer networks in the past decade, the need for security is now bigger than ever. An intrusion detection system (IDS) can add a level of security to a computer network by monitoring all the users in its environment. Generally, an IDS detects attacks by analyzing the payload in messages or commands. Recently, a way of detecting intruders without looking at the contents of a message was introduced [1]. The technique is applied to the specific problem setting of security protocols.

This paper addresses a new and novel technique of gathering meta-information of network messages and describes the technique as implemented. Unfortunately, actual encrypted traffic is not available to test this concept, so our work includes development of a simulation environment. Consequently, our system consists of a monitor, a principal simulation environment, and a security protocol intrusion detection engine. We address our design framework, the software techniques to accomplish the network programming in our environment and how our design relates to the Common Intrusion Detection Framework.

Keywords: Threads, Network Programming, Intrusion Detection, Reference Monitor, Consumer-Producer

1. Introduction

In the 1990's we experienced the dawn of the Internet revolution. Now the Internet is growing at an unprecedented rate and is embedding itself in the fabric of our society. The average American can now trade stocks, check bank accounts, and buy goods online. Unfortunately, this new convenience comes with a price. Network security has not grown in par with the Internet and as a result many Internet users are vulnerable to attacks.

Two approaches to security for electronic communication have emerged: (1) Detection and response and (2) Protection. In the first paradigm, Intrusion Detection Systems (IDS) add a level of security to a computer network by monitoring the users network activity. IDSs build on early security technology that depended on monitoring system logs to determine if malicious activity

occurred. On the other hand, protection mechanisms *prevent* security compromise, usually through access control technology with or without encryption. Encryption has proven to be effective in protecting privacy and enhancing authentication in network communications.

These approaches are each effective mechanisms, though neither is foolproof. Moreover, while they have complementary characteristics, until recently, they have been mutually exclusive in application. There is one issue that is the primary reason for this. Generally, an IDS detects attack by analyzing the payload in messages or commands, while in encrypted environments, payloads are not available for inspection.

Recently, a novel way of detecting intruders without looking at the contents of a message was introduced [1]. This technique proposed analyzing meta-information about packets on the network, specifically targeting characteristics of security protocols. Security protocols are commonly used on networks for authentication purposes, distribution of encryption keys, and initiating and terminating secure communication sessions. These protocols have characteristics that allow detection of known attacks without knowing the content of the message payloads.

This paper discusses an implementation of a monitor, detection engine, and simulation environment that can be used to detect intrusions from encrypted network messages. The monitor system is a client-server application with the server gathering and organizing the meta-information. To test the functionality of the monitor a principal simulation environment was created. This environment can simulate normal, suspicious, and attack behavior.

The rest of this paper is organized as follows: In Section 2, we present background information about intrusion detection systems, and in section 3, summarize the SEADS architecture. In Section 4 we describe the monitor and in section 5, detail the principal simulation environment. Section 6 describes the detection engine while Section 7 details test results. We conclude the paper in Section 7.

2. Intrusion Detection Systems

Numerous intrusion detection systems have been created and applied to a wide range of problems [2]. They can be used on networks to provide an extra layer of security. However, they do not provide security alone. IDSs are designed to complement and assist other forms of security. This interoperability between security systems is essential and represents the time-tested principle of defense in depth.

The Common Intrusion Detection Framework or CIDF is a movement to develop ways to allow intrusion detection engines to interoperate with other programs [5]. One of their attempts is to architecturally divide the IDS into four major independent components that can be reused in other systems: Event Generator, Event Analyzer, Event Database, and the Response Unit

The event generator is the component that samples activity from the network environment and convert the information into objects that can be used by other components. After converting the information into objects, the generator stores the objects in the event Database. The event analyzer retrieves the objects from the event database and analyses them in order to detect intrusions.

There are two main designs available to the event analyzer for detecting attacks: 1) the knowledge-based design and 2) the behavioral-based designs [6]. In theory, an IDS can use either or both design approaches to detect intruders.

Knowledge-based design detects intruders by pattern-matching user activity to known attack signatures. Signatures are kept in a database containing a repertoire of information describing normal, suspicious, or attack behavior. A signature is a description of a behavior. For instance, in an operating system, an attack signature may consist of the following sequence of commands:

su <correct password>
rm –R /*

If the event analyzer detects a sequence of events that matches a corresponding attack signature, then an attack has been detected.

The behavior-based design uses statistical methods or artificial intelligence in order to detect attacks. Profiles of normal activity are created and stored in a database. Any activity gathered by the event generator that deviates from the normal profile in a statistically significant way can be deemed as suspicious activity or an attack.

3. Secure Enclave Attack Detection System (SEADS)

3.1 The Topology of SEADS

SEADS applies the well-known monitor model to an IDS application. SEADS is conceptually divided into three parts similar to the ones described in the CIDF model presented in section 2. The three parts are the Monitor, the Intrusion Detection Engine (IDE), and the Knowledge Base (KB).

The monitor in SEADS is comparable to the CIDF event generator and event database. This is because the monitor gathers information from the network, converts the information to objects and stores these into an internal database. The intrusion detection engine and the knowledge base together are analogous to the event analyzer. The IDE uses the knowledge-base design described in section 2. It retrieves objects from the monitor's database and searches for the presence of attacks by comparing these with signatures stored in the KB. The KB is a repository of normal, suspicious, and attack signatures.

SEADS assumes secure communication between the monitor and principals. Inside this protected environment, the principals can safely forward information to SEADS. The principals communicate between one another via public networks such as the Internet. Accordingly, the intruder only interacts with the principals on the public network.

3.2 The Needham-Schroeder Protocol

In this section, we use a well-known protocol to illustrate how SEADS can detect an attack on a security protocol. The Needham-Schroeder Protocol (NSP) is a popular and widely used key distribution and authentication protocol. This protocol was first introduced in [12] in 1978 and now countless papers show how intruders can spoof the participants by replaying messages. The protocol contains the messages shown in Table 1.

1.	A -> S:	A,B,na
2.	S -> A:	{na,B,kab,{kab,A}kbs}kas
3.	A -> B:	{kab,A}kbs
4.	B -> A:	{nb}kab
5.	A -> B:	{nb-1}kab

Table 1

The NSP protocol consists of five messages and involves the participation of three parties. Since the messages are encrypted, their contents cannot be used to detect attacks. A primary contribution of the work on SEADS is that there is other pertinent information available. For instance, every message in the NSP protocol is sent by one participant and received by another. The series of send and received events are valuable information that does not involve the decryption of messages. The NSP protocol is shown in Table 2 as a series of send and receive events.

We now show the Denning and Sacco attack on NSP [3] and how it is detected in this architecture. The attack requires the intruder to intercept messages from one session, compromise a session key, and open a second session to replay the intercepted messages.

Effectively, the attack is enacted by the intruder replaying message #3 from the compromised session to the same recipient that originally received the message. Even though payloads are encrypted in NSP messages, the intruder is able to obtain authentication from B. The intruder does not have to decipher the payloads in order to perform this attack. Instead, the intruder relies on copying and replaying messages.

In order for this attack to be possible, the intruder needs to be sophisticated enough to remove and insert messages in the network at will. Unfortunately, the technology to do this is available to many intruders.

The events that identify the attack are given in Table 3. When the IDS detects these three events signified by the action, protocol message number and session, it should signal that an attack has occurred.

Seq #	Action	Protocol Msg #
1.	A -> S	1
2.	S <- A	1
3.	S -> A	2
4.	A <- S	2
5.	A -> B	3
6.	B <- A	3
7.	B -> A	4
8.	A <- B	4
9.	A -> B	5
10.	B <- A	5

Table 2

Seq #	Action	Protocol Msg #	Session ID
1.	B<-A	3	x
2.	B->A	4	x
3.	B <- A:	5	x

Table 3

4. The Monitor

4.1 The Monitor Database

In an intrusion detection system, the monitor is the component that gathers traffic between principals and other pertinent activity. It packages this information into events and stores them in an internal database for later use or forwards them directly to the intrusion detection engine.

Our monitor is novel because it gathers information without looking at the contents of the network traffic. The information that is collected is meta-information about the traffic.

Specifically, we utilize characteristics about security protocols and attacks gleaned from years of formal method research.

In order to gather the necessary meta-information, principals are required to report events to the monitor. You may recall from section 3, that during the execution of a protocol, a series of messages are exchanged between principals. Each message in the protocol consist of at least two events: a send and a receive event. These send and receive events are the ones forwarded to the monitor. Thus, the principals execute cooperating processes that automatically communicate with the monitor. Every time a protocol message is sent, the principal notifies the monitor by reporting it as an event; the same for receive events.

For our implementation of events, we selected a minimal set of data that we can use to identify attacks. These include the identity of the acting principal (PN), a session identifier (nonce), other parties to the session (Parties), and the type of event (send or receive).

A session represents one execution of a security protocol. At any given time, the monitor can be gathering information from countless sessions involving different principals. It is crucial for the monitor to efficiently record the event and store it in its database. Figure 4a shows how an event is stored

As shown in Figure 4.1, a session can be distinguished from any other session with its PN, Parties and Nonce fields. Each session in turn has a collection of events.

It is worth noting that this organization of events by the monitor aids the intrusion detection engine in detecting attacks. This is due because many known attacks span multiple sessions involving the same group of principals [4]. Since the monitor's database stores events according to the group of principals involved, it is easy and fast for the IDE to retrieve this information.

4.2 The Monitor's Threads

The monitor was designed to be robust and able to handle a high volume of sessions. To accomplish this, a multi-threaded design was chosen, commonly referred to as the consumer-producer thread design.

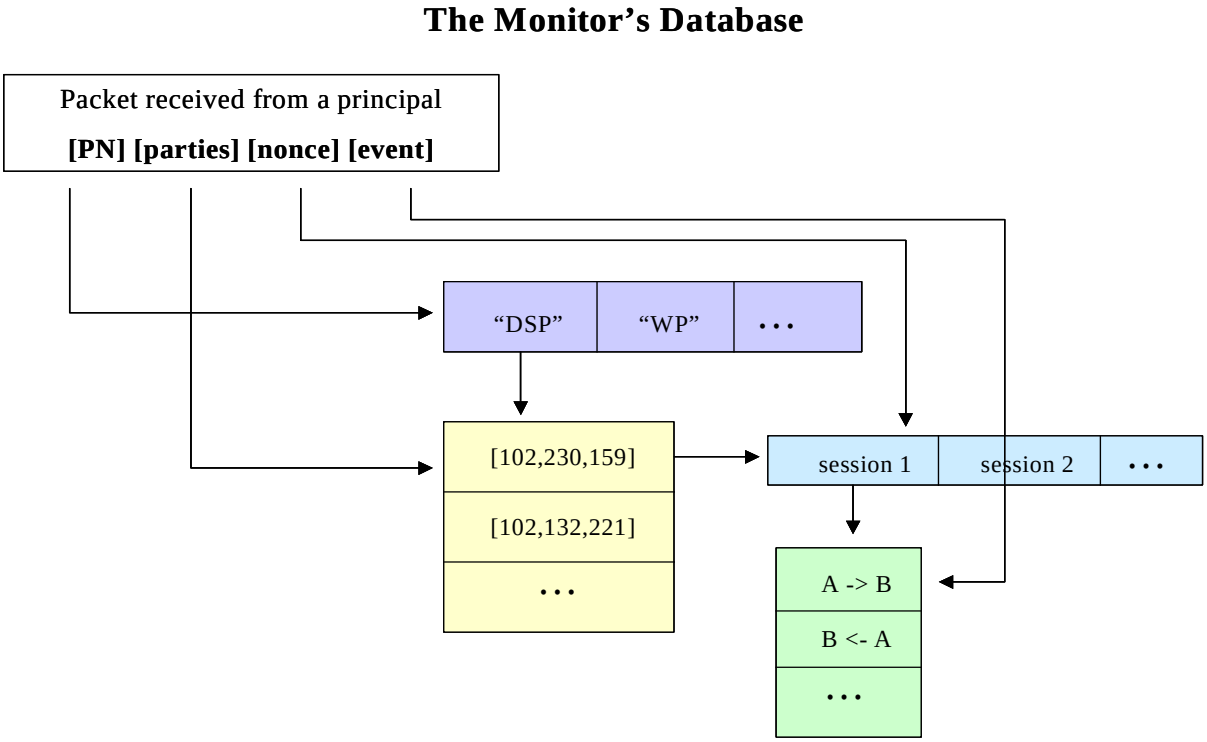

Figure 4.1

In this case, the consumer is a thread, which is constantly listening to network socket connections and managing all the open sockets. Any information the consumer reads from a socket is quickly placed in a queue. The producer thread takes the packets waiting in the queue, checks them for proper format, converts them to event objects and then stores the objects in the monitor's database.

After storing an object in the database, the producer thread signals the IDE engine about the presence of new events. In turn, the IDE uses a well-defined interface provided to it by the monitor to retrieve events from the monitor's database. This consumer-producer thread design helps the monitor handle many concurrent sessions by shifting the bottleneck from the network socket's I/O and into the internal, dynamic queue of the monitor. Figure 4.2 illustrates the monitor's threads collaborating in a consumer-producer thread design.

4.3 The Monitor's Code

The monitor was coded in Visual C++ for the Win32 platform. Win32 kernel objects such as sockets, threads, events and critical sections [7] were used. The sockets allowed the monitor to listen for network traffic and the threads were used in the coding of the consumer-producer monitor design.

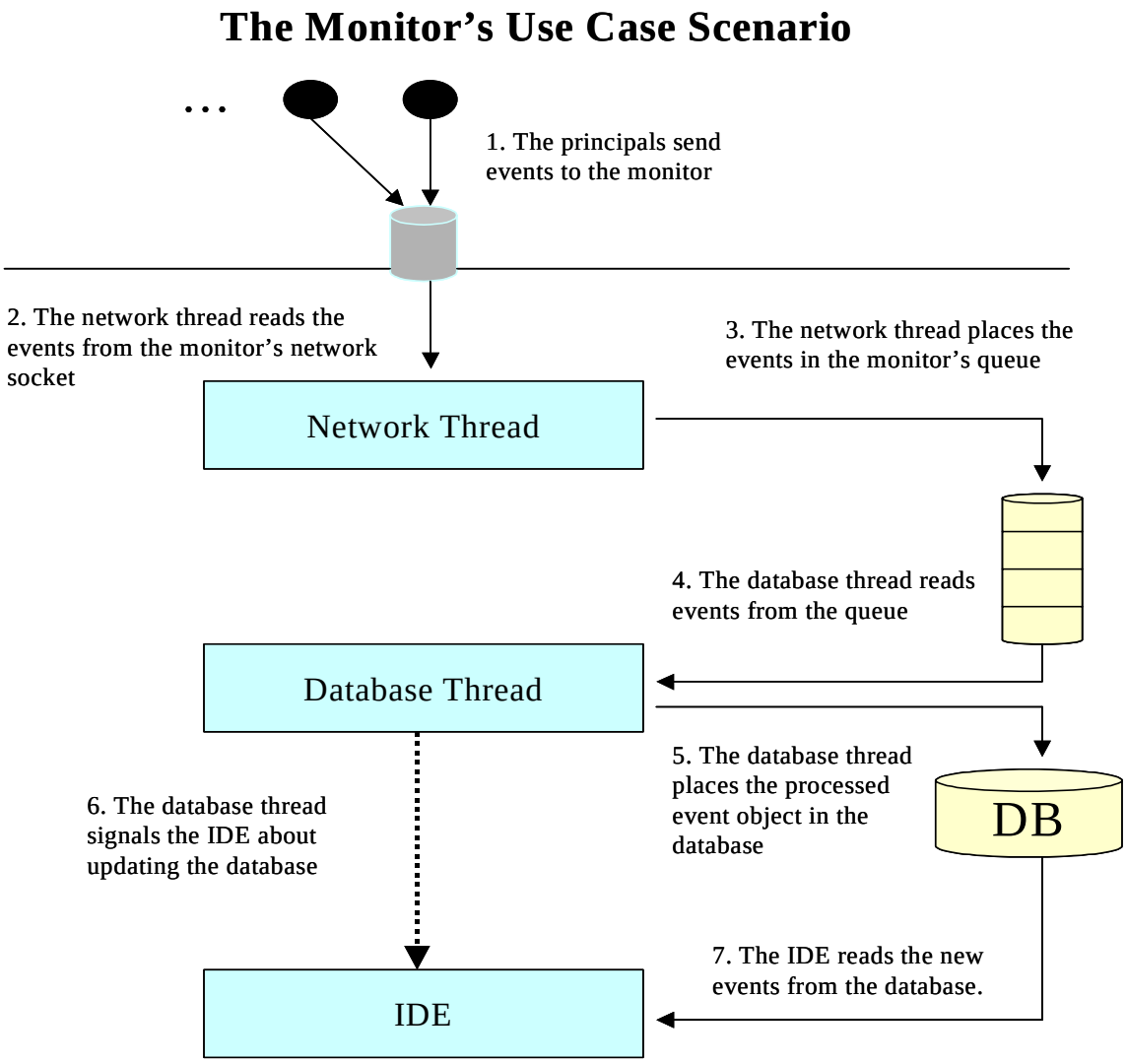

Figure 4.2

Standard Template Library [8] containers were also used. For instance, the monitor's database was created with maps, linked-lists, and vectors. Since these containers grow dynamically, the monitor's database can hold as much data as possible limited only by the computer's memory.

5. The Principal Simulation Environment

5.1 The Components of the Principal Simulation Environment

In order to test the functionality and correctness of the monitor, a network environment of principals was developed. The principals were created with the intelligence to initiate and engage in security protocol sessions involving other autonomous principals. The principals can run on any Windows computer and execute any given protocol signature over the network. During a session, the principals report the completion of events to the monitor. The principals have the ability to engage in normal, suspicious or attack behavior.

The Principal Simulation Environment is divided into three different programs: Principal Simulator, Principal Dispatcher, and The Principals themselves. The Principal Simulator provides the user-interface for creating an environment of principals. Each simulation requires the input of parameters that are used to configure the system. Some of the configuration parameters that the user can customize are the number of sessions to be executed, the computers involved in the simulation and the protocols and signatures that the principals will execute.

After the simulation has been created, it is the job of the Principal Dispatcher to instantiate the principals at a given computer when instructed to do so by the Principal Simulator. The autonomous principals then communicate with each other, execute protocols and report events to the monitor. Figure 5.1 illustrates how the three components interact to produce a simulation.

5.2 The Principal Simulator

The Principal Simulator is the program that configures the principal's network environment. This is a Graphical User Interface program that provides the user with an easy to use interface to create and run simulations.

All the commands necessary to work with the simulator are presented as menu items in the menu bar. The toolbar also contains the most commonly used commands such as run, new, edit, save and print. The user can create a new simulation by clicking on the new command. This command will show a dialog box that permits the user to add activities to the simulation. A simulation consists of activities and each activity requires configuration parameters for the protocol name, number of sessions, identity of principals, signatures to use, and a start time for the simulation.

In order to assist the user in selecting a protocol name, the program reads the file containing the protocol signatures and populates the “Protocol Name” combo box with the available protocols.

The Principal Simulation Environment’s Use Case

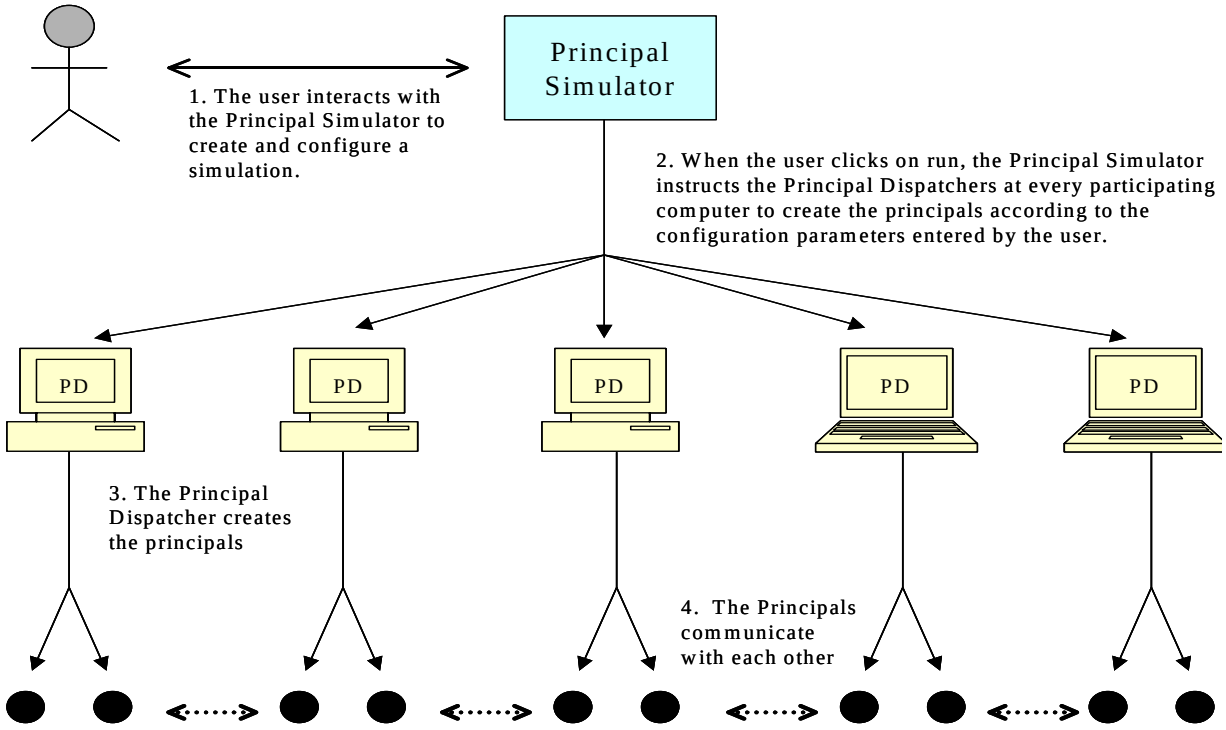

Figure 5.1

Again the program reads the protocol signature file to populate another combo box listing the attack signatures available for this particular protocol. The attack signatures are numbered from 0 to (n-1) where n is the total number of signatures given for the protocol. Once the user has finished adding activity to the simulation, all the configuration parameters are printed to the screen as shown in Figure 5.2.

After the simulation is created, the user has the choice of editing, saving, printing deleting or the simulation. When the user clicks on run, the Principal Simulator communicates with every Principal Dispatcher running on all the participating computers. The Principal Dispatcher in turns creates the principals that will run on his computer.

5.3 The Principal Dispatcher

The reason for having a Principal Dispatcher has to do with the inability of creating processes on a computer remotely. This inability is expected since the ability to start remote processes on a computer can be seen as a security breach.

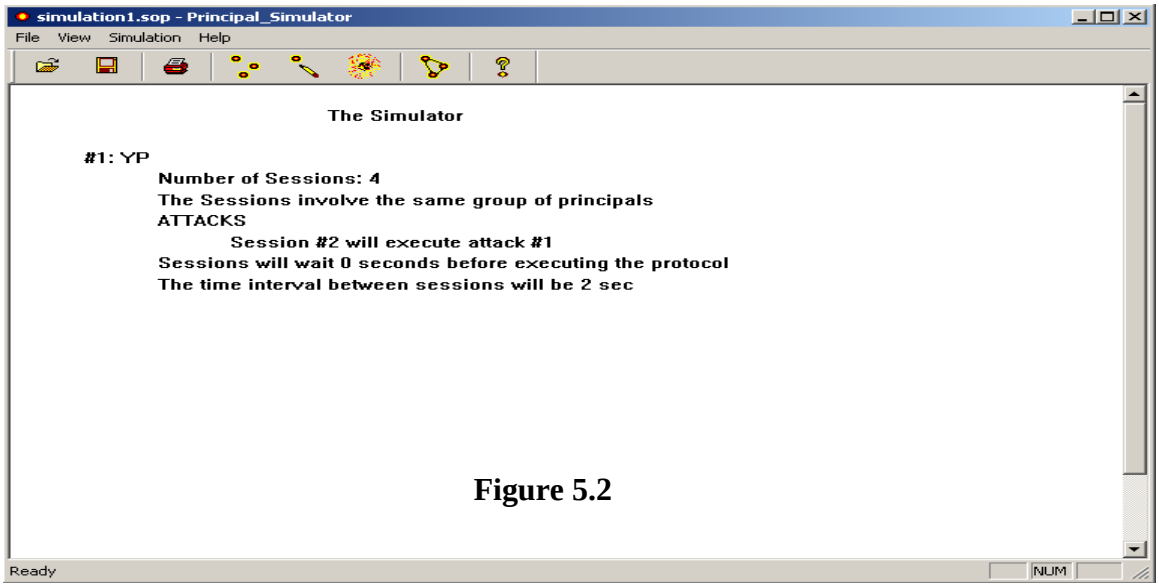

Figure 5.2

One solution to this problem is to create a process in every computer that listens to the network on a pre-established port number. The Principal Simulator sends instructions to each Principal Dispatcher at this port number. The instructions contain the number of Principals to be created and configuration parameters for each. After getting the instructions, the dispatcher creates each Principal.

5.4 The Principals

Principals are autonomous network programs that engage in sessions with other principals. They execute protocols and report events to the monitor. The design issue for this program was figuring out the easiest way to create a principal that could engage in normal and attack behavior when instructed? The answer was to provide a file containing signatures of normal, suspicious and attack behavior that the principals could read and execute.

This file, named “Simulation_File.txt”, is almost identical to the one provided to the intrusion detection engine by the knowledge base. The file is divided by protocols and each protocol contains at most one normal signature. Any additional signatures in the file represent suspicious

or attack scenarios. An example of a signature present in the file is the one shown below executing a normal session of the Denning-Sacco Protocol or DSP:

When the Principal Dispatcher creates the Principal, the first step is to read "Simulation_File.txt". After picking the selected signature from the database the Principal determines the number of other Principals involved. This number may differ since different security protocols differ in the number of participating entities. All protocols involve at least two principals.

The next step is to determine the initiating principal. The initiator is responsible for creating a random number called a nonce that is used to identify the session. As you may recall, the monitor uniquely identifies sessions by the protocol name, group of principals and nonce. The initiating principal is now ready to send the first message to the corresponding party. During execution of the protocol signature, the principals report their activities to the monitor. Figure 5.3 shows the flow of the Principal program.

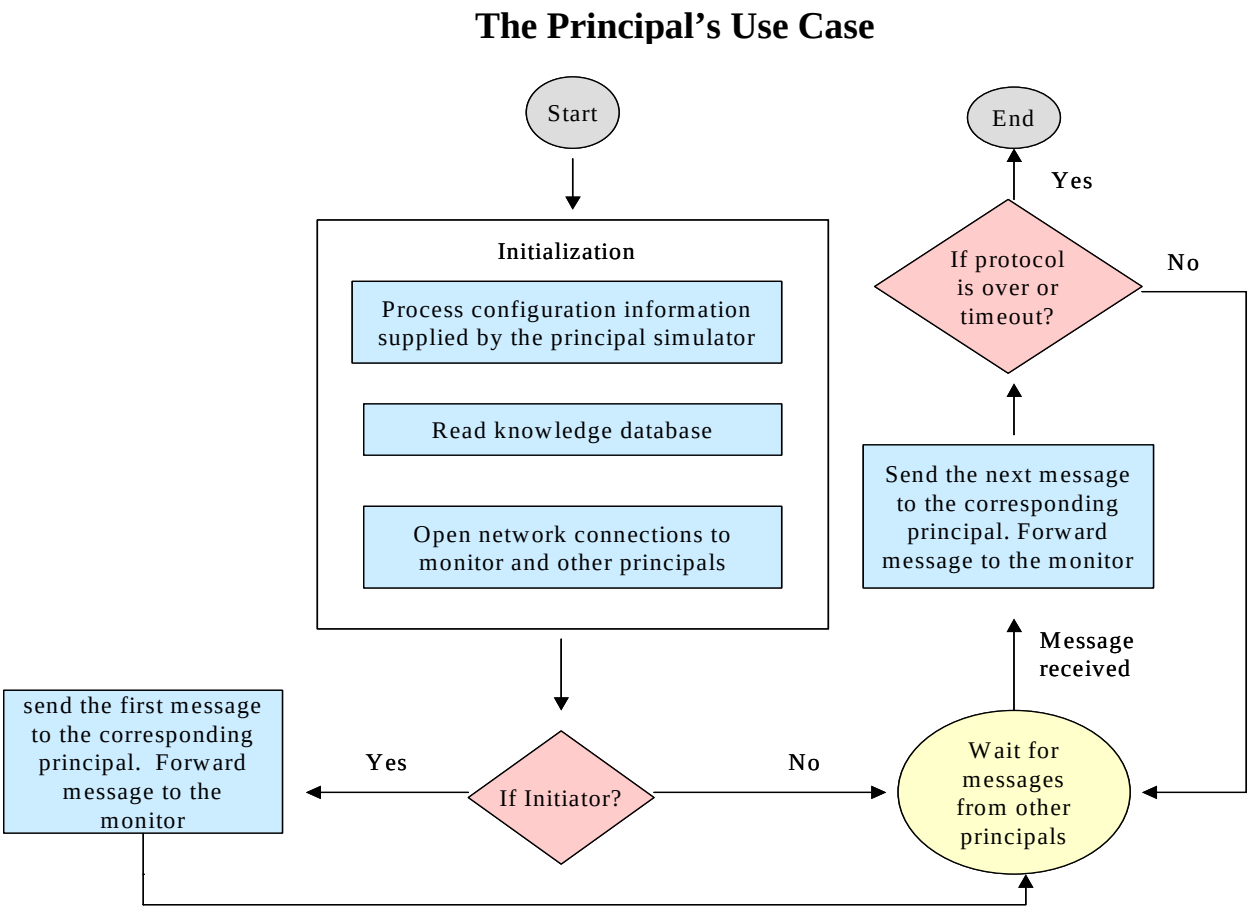

Figure 5.3

The principals have the special ability to open multiple sessions with the same group of principals. They can accomplish this feat with the assistance of threads. This feature is necessary since some protocols require the execution of parallel sessions. In addition, opening multiple sessions allows for the simulation of particular sophisticated attacks. These types of attacks usually involve the intruder opening multiple sessions with the same group of principals.

The IDE uses distinct detection methodologies for protocol attacks depending on the number of sessions used in each specific attack. Attacks on security protocols may be over only a single session of the protocol or may utilize information gleaned from multiple runs of the protocol. Thus, attacks may be classified as *Single session* attacks or *Multi-session* attacks.

Single session attacks are those attacks which may occur in a single session. The signature of such an attack may differ from the protocol itself in only something so subtle as a missing receive statement. In our environment, these subtle differences are easily recognized.

Interestingly, we consider the attack on the Needham and Schroeder Conventional Key Protocol (NSCKP) a single session attack even though the attack depends on a previously compromised key from another session. The telling factor is that the attack can be detected by recognition of a single protocol session. In the NSCKP case, even though it is, technically a replay attack, it can be recognized by the signature given in Table 1 without any knowledge of the previous session.

Detection of single session attacks by the IDE is simply a matter of the relevant attack finite state machine reaching the final state, upon which the IDE will signal a notification. No knowledge of the previous session is necessary for the IDE to detect this attack.

Multi-session attacks are those attacks that use information extracted from more than one previous or concurrent protocol sessions. We make the reasonable assumption that such attack sessions must use the information within a certain time period of the reference session(s), from which the information is taken in order to subvert the protocol. For multi-session attacks, the IDE classifies them as either Replay Attacks or Parallel Session Attacks.

Replay attacks use information extracted from a previous run of a protocol. The first question that must be answered is: "How much time can pass between the reference session and the attack session?" This is an important question in our architecture because of the way replay attacks are detected. The signature of a replay attack consists of the signature of the reference session

followed by the signature of the attack session. Thus, the recognizer must remain active until either an attack is detected or the threshold period expires.

We handle this by requiring the author of signatures of replay attacks to include the threshold in the signature, which will vary from protocol to protocol. The default wait constant was chosen to be ten seconds for the IDE prototype. If events occur that triggers a replay recognizer, if the time difference between the attack session and the reference session is greater than the wait time, the IDE will flag this activity as suspicious behavior.

A parallel session attack occurs when two or more protocol runs are executed concurrently and messages from one run (the reference session) are used to form spoofed messages in another run (the attack session). As a simple example consider the following One-Way Authentication Protocol (OWAP) [13]:

A → B : E(Kab : Na)

B → A : E(Kab : Na + 1)

Successful execution should convince A that B is operational since only B could have formed the appropriate response to the challenge issued in the first message. An intruder can play the role of B both as responder and initiator. The attack works by starting another protocol run in response to the initial challenge.

To initiate the attack, Mallory waits for Alice to initiate the first protocol session with Bob. Mallory intercepts the message and pretends to be Bob, starting the second run of the protocol by replaying the intercepted message. Alice replies to Mallory's challenge with exactly the value that Mallory requires to accurately complete the attack session. The attack is shown in Figure 5.4.

Attack Session	Reference Session
A → M(B): E(Kab : Na)	
	M(B) → A: E(Kab : Na)
	A → M(B): E(Kab :Na + 1)
M(B) → A: E(Kab : Na + 1)	

Figure 5.4

The IDE detects parallel session attacks by matching the ongoing activity against the attack signatures. The telling factor in this case is the omission of any information from Alice's partners in either session, as reflected in the signature in Table 4.

Current State	Event	Protocol	Session	Sender	Receiver	Message Number	Next State
SS	send	OWAP	X	A	B	1	S1
S1	receive	OWAP	X+α	B	A	1	S2
S2	send	OWAP	X+α	**A**	**B**	2	**S3**
S3	receive	OWAP	X	**B**	**A**	2	FS

Table 4

6. Intrusion Detection Engine Design

This section provides an insight into the design of the Intrusion Detection Engine. Justification of the major design decisions is also given. The design of the IDE uses the object-oriented paradigm. The problem was broken down into smaller components, and appropriate classes were developed to accurately represent the problem.

A major factor in the design of the IDE, was the complexity of the environment being monitored. Within any enclave, we expect to monitor events interleaved from multiple:

- Concurrent sessions
- Different principals
- Different protocols

In addition there is no guarantee that all the sessions will properly conclude. Some sessions may be suspended abnormally and messages may be lost.

a. Architectural Design

A number of issues had to be taken into account in the design phase of this research implementation. The design was created in order to ensure that all the requirements and specifications were satisfied.

In the secure enclave it is possible to have multiple concurrent sessions of different protocols executing within the enclave. The sessions may consist of the same or different principals. The Intrusion detection engine must be able to keep track of the different protocol sessions executing within the enclave in order to detect any attacks or suspicious activity. Not all attacks on security protocols occur over a single session. As described earlier, multi-session attacks such as replay attacks or parallel attacks may occur within the enclave. These multi-session attacks span multiple different protocol sessions. The Intrusion detection engine must provide a means to keep track of such executing sessions and detect any attacks.

Additionally, the detection of attacks has to be communicated to the person or system monitoring the enclave. Detailed reports of all attacks or suspicious behavior must be generated by the IDE. Such reports provide in-depth information about the type of attack and principals participating in the protocol session. The Intrusion Detection Engine receives crucial inputs from the Activity Monitor and from the Knowledge base of protocol signatures. It is important to ensure that interfaces with the Monitor and the Knowledge base are well defined and reliable.

b. The Thread Dispatcher and Monitors

As noted earlier, the IDE receives protocol events from the monitor as they occur. The IDE is multi-threaded with a single thread to serve as the thread dispatcher. Since each protocol may have many attack signatures associated with it, when a new protocol session begins, the IDE spawns a new thread to monitor all the FSM recognizers for that protocol. As illustrated in Figure 6.1, the Thread Dispatcher then routes events to the appropriate thread as they arrive.

To keep track of all the threads existing within the system, a *ThreadList* class is employed,

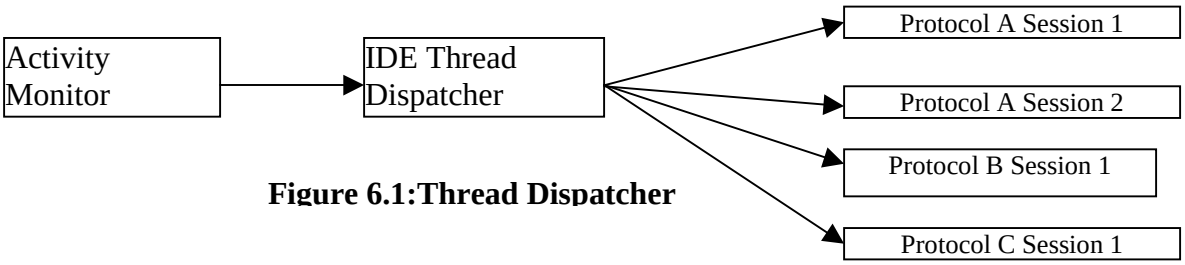

Figure 6.1:Thread Dispatcher

that holds the protocol name, session number, identifiers of the principals involved, a signal to which the thread listens, and a thread identifier for each thread.

The threads provide the detailed functionality of the Intrusion Detection Engine. Each thread monitors the activity within a single protocol session. As events for a particular protocol session come in from the activity monitor, the thread matches those events against the protocol signatures stored in the knowledge base. If a event matches, the Finite State Machine corresponding to that particular signature is advanced to the next state.

Upon conclusion of an attack session or a normal protocol session, it may so happen that the entire signature from the knowledge base matches the succession of events for that protocol session coming in from the activity monitor. In such cases, the thread will raise alerts to the console, providing information about the attack or normal session. If an attack is detected by the Intrusion Detection Engine, the detailed information about that attack is written to a text file.

This information is used by the Graphical User Interface component of the IDE to generate the attack reports.

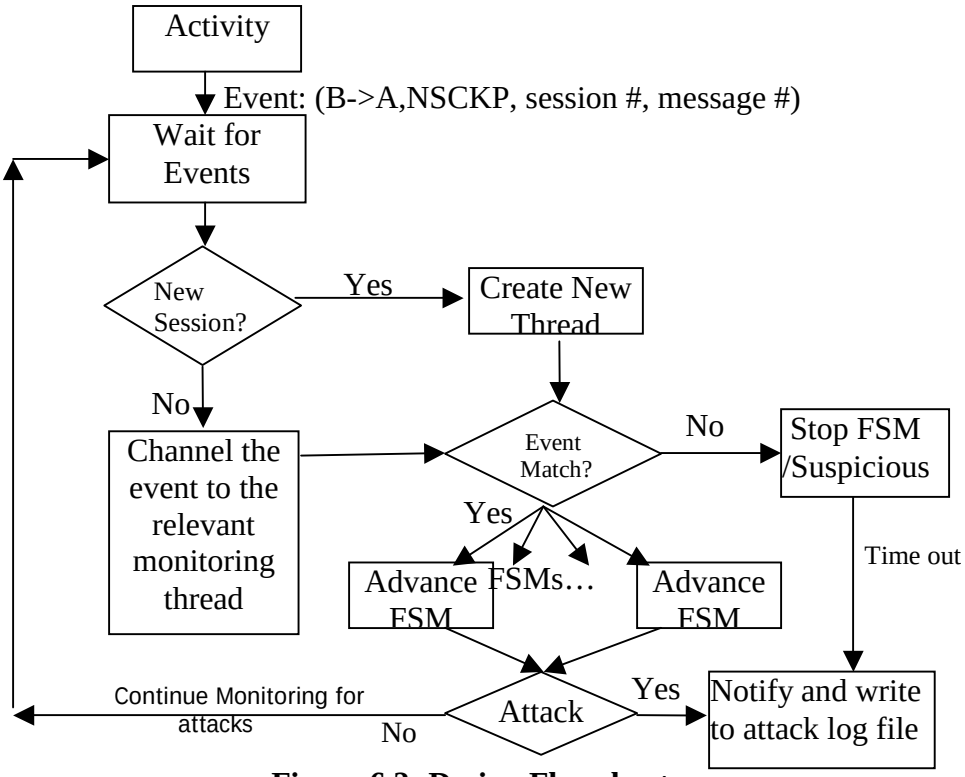

Figure 6.2: Design Flowchart.

Threads terminate in two normal ways: (1) An attack is detected, or (2) The protocol ends normally. However if a particular protocol session hangs with no further events coming into the IDE, the thread will die after a timeout period and it will signal the activity as an abnormal termination. When a thread dies, the corresponding entry from the list of threads designed as an object of the ThreadList class is removed.

Threads are chosen as control structure of choice for the IDE for several reasons. First, the number of concurrent threads spawned by a process is limited only by the virtual memory on the system. This allows the IDE to track a large number of concurrent sessions, accurately representing an Internet environment that is rich with security protocols. Secondly, there are no synchronization issues to be taken care of as all the threads have their own memory space and can also access the global variables. Any data structure that is accessed by all the threads has been protected by means of a critical section. The overall design of the IDE is reflected in the flow chart in Figure 6.2.

7. Test and Results

Upon completion of each significant milestone, the IDE was tested to ensure that the product functioned correctly. We approached the testing from four standpoints:

(1) Detection of attacks against protocols in all three categories of single session, replay, and parallel session
(2) Detection of suspicious activity
(3) Effective operation in a highly concurrent environment
(4) Effective user interface.

We began our testing by addressing the ability of the IDE to detect different categories of attacks. The environment being monitored was systematically subjected to attacks of each of the three categories of single session, replay and parallel session. For the single session attacks we simulated those attacks on protocols which span over only a single session. The single session attack on the Needham and Schroeder Conventional Key Protocol (NSCKP) explained in detail earlier was one of the many attacks that were simulated. The IDE was correctly able to detect all such single session attacks.

To test the replay attacks, we simulated a correct run of protocols such as the Ottway-Rees Protocol (ORP) [14] and, within 10 seconds, we ran an attack session on the same protocol. In every instance, the IDE detected such replay attacks and classified them correctly.

To test our ability to detect parallel session attacks, we ran the parallel session attack on the Woo and Lam Authentication Protocol First (WLAPF), which was successfully detected by the IDE. We also ensured that protocol activity which may be considered abnormal or suspicious was detected by the IDE. Event sequences not corresponding to any attacks currently existing in the knowledge base or normal protocol runs were simulated for protocols. The IDE was correctly able to report such activity as unrecognizable suspicious activity on the basis of its inability to find a complete match for that particular signature in the Knowledge base. It is not always the case that protocol sessions successfully run to termination. Events get lost or the protocol session may stall. We simulated a protocol session in which there is abnormal termination before the current run has reached its completion In such cases the IDE thread monitoring this session times out after the TIMEOUT period and reports abnormal termination of the protocol.

It was important to ensure that the IDE is able to function correctly under a highly concurrent environment. *Sixty* concurrent sessions of different security protocols were simulated. These

included attack sessions as well as correct sessions. Specifically, five distinct protocols were executed. A total of one hundred seventy principals were concurrently executing within the enclave. Out of the sixty protocol sessions, forty sessions were attack sessions and twenty sessions were normal protocol sessions. The IDE was able to correctly detect attacks and report them to the Graphical User Interface.

The Graphical User Interface is an integral part of our research implementation. This GUI allows the user to have an overall detailed view of all the attacks that took place within the environment over any given period of time. After each attack is detected, the IDE writes the detailed attack report to an attack log file. This attack report file is used by the GUI to provide the user with customized attack reports. We tested the functionality of the GUI after each simulated attack was detected to ensure that the attack has been logged and its details are displayed by the GUI. Moreover, we ensured that on providing inputs to the GUI it will only display the attack reports for specific protocols over a specific duration of time.

Based on the results obtained from the numerous tests performed on the IDE we can say that the IDE interfaces correctly and seamlessly with the activity monitor and the knowledge base. During the correct functioning of the IDE, there is no loss of events between the IDE and the monitor and hence no loss of functionality of one due to the other. Also, signatures can be added to the Knowledge to allow the IDE to detect the additional attacks on protocols.

Extensive testing on the IDE shows that the IDE fulfills its functionality successfully. The IDE can be used to detect different types of attacks on security protocols under environments of high concurrency. The Graphical User Interface also proved to be very reliable in order to increase the amount of information available to the user upon occurrence of such attacks.

Extensive testing and demos were conducted to test the functionality of the Monitor and the Principal Simulation Environment. We initially conducted limited tests to ensure that a single session could be recorded. We gradually increased the workload, varying the size and nature of the traffic. As example, one test included eleven sessions of three different protocols, with twenty four different principals participating. Another simulation exercised one hundred and twenty five concurrent sessions.

We also exercised sessions that modeled classic replay attacks as well as more complex parallel session attacks. In all the tests, the software executed according to specifications. Stress

tests were also conducted to test the robustness of the software. These tests primarily involved overloading the network with a multitude of sessions executing different protocols and involving different principals. The Principals were successful at generating a large volume of traffic and the monitor was able to gather all event information from the principals.

8. Conclusion

This monitor program shows that relevant and useful information can be gathered without having to examine the payload of messages exchanged between principals. This is the first instance that we are aware of where security protocols have been analyzed in an environment comprised of different protocols running multiple concurrent sessions with multiple users. This is particularly significant because of the importance of encryption in protecting networks and computers in the future.

An integral part of this work was the creation of the Principal Simulation Environment. The monitor needs the active participation of the principals in order to collect the meta-information from the network traffic. The principals are autonomous network programs that execute signatures between each other and report the events to the monitor.

9. Bibliography

[1] Alec Yasinsac, "An Environment for Security Protocol Intrusion Detection", Journal of Computer Security, Vol. 10, pp. 177-88, No. 1-2, 2002

[2] Alec Yasinsac, "Active Protection of Trusted Security Services", Technical Report TR--000101, Department of Computer Science, Florida State University, Jan 2000

[3] D. E. Denning and G. M. Sacco, "Timestamps in key distribution protocols," Communications of the ACM, vol. 24, no. 8, Aug 1981, pp. 533-536

[4] John Clark and Jeremy Jacob, “A Survey of Authentication Protocol Literature: Version 1.0”,A continually updated library of protocols analyzed in the literature, available at www.cs.york.ac.uk/~jac/, 1997

[5] Brian Tung, *Common Intrusion Detection Framework (CIDF)-website*, www.gidos.org

[6] Dorothy E. Denning, “An Intrusion-Detection Model”, From 1986 IEEE Computer Society Symposium on Research in Security and Privacy, pp118-131

[7] Aaron Cohen and Mike Woodring, *Win32 Multithreaded Programming*, O’Reilly Press, 1998

[8] Nicolai M. Josuttis, *The C++ Standard Library*, Addison-Wesley, 1999

[9] Chuck Sphar, *Learn Microsoft Visual C++ 6.0 Now*, Microsoft Press, 1999

[10] Anthony Jones, *Network Programming for Microsoft Windows*, Microsoft Press, 1999

[11] Robert C. Martin, “UML Tutorial”, www.uml.org, Nov. 1998

[12] Roger M. Needham, Michael D. Schroeder, “Using Encryption for Authentication in Large Networks of Computers”, Comm of the ACM, December 1978 vol. 21, #12, pp.993-999

[13] John Clark & Jeremy Jacob, “Attacking Authentication Protocols”, High Integrity Systems 1(5):465-474, August 1996.

[14] Otwy, D., and Rees, O. 'Efficient and timely mutual authentication'. Operating Systems Review 21, 1(Jan. 1987), pp. 8-10