Florida State University
College of Arts and Sciences
Department of Computer Science

A Method for Detecting Intrusions on Encrypted Traffic

-Sachin Prakash Goregaoker

Project submitted to the Department of Computer Science
in partial fulfillment of the requirements for the
Master of Science degree (M.S.)

The members of the committee approve the Master's Project of Sachin Goregaoker defended on July 18th, 2001.

Dr. Alec Yasinsac
Major Professor

Dr. Robert van Engelen
Committee Member

Dr. Lois Hawkes
Committee Member

Acknowledgements

The completion of this Master's project would not have been possible without the timely guidance and encouragement of my advisor, Dr. Alec Yasinsac. It was due to him that I gained a better understanding of a field of computer science, which is relatively new to me. I would also like to thank my parents who are a constant source of inspiration and support to me. My brother, Sameer has also been supportive of me through my tough years completing the Master's degree. Lastly, but not least I'd like to thank the other members of my research group, Nikhil Patel and Alex Melendez. I had the privilege of working closely with them during my research, and have found them to be great colleagues.

Table of Contents

1. Introduction

In this day and age of unprecedented growth in the fields of web-communication and more specifically, e-commerce, it is all the more important to ensure secure communication between all parties involved. Web sites routinely establish transactions over the Internet with their customers and in the process, credit card numbers and other sensitive information about the customers, is sent. Ensuring the integrity and security of such information has spawned a huge amount of research in the fields of network security and cryptography. Intrusion detection systems aim to detect attacks against vulnerable computer systems and networks. In this paper, we show how Intrusion Detection technology can be used in encrypted environments.

1.1. Need for Encryption

Any data exchanges over public networks are susceptible to interception by malicious parties. Unless cryptography is used in order to encrypt such exchanges, data is vulnerable to passive listeners. Secure electronic communication relies on the application of cryptography. A number of cryptographic algorithms exist to allow the involved parties to ensure the level of data security that they need.

Generally algorithms such as DES (Data Encryption Standard), IDEA (International Data Encryption Algorithm) and a host of others are used to encrypt the data. In addition to the encryption algorithm the valid parties involved in the communication need to possess the valid keys to encrypt and decrypt the information. A combination of the cryptographic algorithm and encryption keys will result in an encrypted exchange of information between the valid parties involved.

However even perfect encryption may not be enough to prevent communication from being compromised. There have to be certain rules or protocols, which govern the encrypted exchanges, to allow for key exchange, authentication and privacy.

1.2. Flaws in Security Protocols

Encryption backed by effective security protocols may still not be enough to prevent sophisticated intruders from compromising the so-called secure communication. Even if encryption is used to protect the data, there still exist flaws in security protocols, which can be used by malicious parties to their benefit, so as to subvert the underlying protocol.

Intruders can still spoof the valid parties in any data exchange by using replay techniques, whereby information from previous runs of any encrypted exchanges are used in the current run. As a result, intruders may be able to masquerade as valid parties in communication, steal keys etc. all of which leads to the inevitable compromise of the encrypted exchange. It is clear that another level of protection must be provided for encrypted data exchanges to detect attacks on the security protocols.

1.3. Detecting Attacks on Security Protocols

The future of network security revolves around *secure enclaves*. Secure enclaves are environments where every, and only, recognized enclave members operate securely within the environment. Environments where electronic communication is protected by cryptography are secure enclaves as long as the members of the enclave are protected from data loss, compromise and malicious attacks. In this project, I demonstrate a technique to detect attacks on encrypted exchanges between parties in such a secure enclave. The Intrusion Detection Engine is able to raise alerts upon detection of positive attacks or suspicious activity within the secure enclave.

1.4. Terminology

Principal: A process running on a computer.

Enclave: A group of valid principals operating within a secure environment.

Intrusion Detection Engine (IDE): The working prototype developed as a result of this project.

SEADS: Secure Enclave Attack Detection System.

Knowledge Base: The database of attack signatures used by the IDE as a reference.

Activity Monitor: The monitoring component of SEADS, which gathers ongoing protocol activity dynamically as it happens within the secure enclave.

Signature: The trace of events, which correspond to a run of a single protocol session. Signatures may however correspond to correct protocol runs or protocol attacks.

2. Intrusion Detection Overview

The aim of intrusion detection systems is to detect attacks against computer systems and networks. There has been growing interest in this particular field of computer security. The research therein has yielded various methodologies of intrusion detection. The task of intrusion detection systems is to monitor the usage of information systems and detect the appearance of insecure states. Intrusion detection systems detect attempts by legitimate users of the information systems or external parties, to abuse their privileges or to exploit security vulnerabilities. The widespread research on intrusion detection systems is due to the difficulty of ensuring that an information system will be free of security flaws.

Since the seminal work by Denning [3], many intrusion detection models and prototypes have been created. The most generic classification of intrusion-detection systems can be based upon the model of detection. An intrusion detection system which detects malicious activity/ intrusions based upon the behavior of the attacker uses the *Behavior-based* or *Anomaly Detection* model of intrusion detection. An intrusion detection system which uses a knowledge base of known attacks on the system being monitored or on the security protocols being used for communication, follows the *Knowledge-Based* or *Misuse Detection* model of intrusion detection.

2.1. Behavior Based Intrusion Detection

Behavior based intrusion detection systems use the information concerning the behavior of users within the monitored environment. If any user exhibits unrecognized behavior, then the intrusion detection engine will generate an alarm. Behavior based intrusion detection systems will base their detection mechanism on deviations from the expected or normal behavior of users.

2.1.1. Behavior Based Detection Models

Some important models for behavior based intrusion detection are:

- Statistics

- Expert Systems
- Neural Networks

Statistics

Statistical modeling [2] is a widely used tool for behavior based intrusion detection systems. Over a period of time, user behavior is gathered and stored as profile information. The time period may vary from a few hours to a few months, depending on the profiling needs. Profile information may include *login frequency, resource use, session duration* etc. User profiles are dynamically updated so that as user behavior changes so do their profiles. This model has been widely used in various tools for intrusion detection.

Expert Systems

Expert systems can be used for behavior based intrusion detection. In [3] the author describes a prototype real time Intrusion Detection Expert System (IDES). This prototype monitors users on a remote system, using audit records that characterize their activities. As it is an expert system, it learns the normal behavior of each user and reports any anomalous behavior.

The IDES monitors target system activity as it is recorded in audit records generated by the target system. IDES examines the audit records as they are received from the target system and ascertains whether the observed activity is abnormal with respect to the user profile. The profiles themselves change dynamically on the basis of observed activity.

Neural Networks

Neural network algorithms [2] are emerging as a new artificial intelligence technique that can be applied to real-life problems. The use of artificial intelligence for detecting intrusions on computer systems is now widely considered as the only way to build efficient and adaptive intrusion detection systems. Among the main uses of neural networks for intrusion detection is to learn the behavior of actors (i.e. users, daemons) in the system. However neural networks are still a computationally intensive technique and are not widely used in intrusion detection.

2.2. Knowledge Based Intrusion Detection

Knowledge Based Intrusion Detection techniques base their detection mechanism on a database comprised of specific attacks and system vulnerabilities. The intrusion detection system contains information about these vulnerabilities and looks for attempts to exploit them. When such an attempt by a malicious party to compromise the security of the system is detected, an alarm is triggered. Knowledge based intrusion detection is also known as misuse detection or detection by appearance, since it depends on recognizing a known pattern of malicious activity.

2.2.1. Signature Analysis Using State Transition Diagrams

In this project, the intrusion detection methodology revolves around the knowledge-based paradigm. The actual detection of attacks on the encrypted exchanges is done by using state transition analysis of attack signatures, stored in the knowledge base. This approach to knowledge based intrusion detection depends on the representation of attacks, on the encrypted exchanges between principals, as signatures, which can be modeled as finite state machines. Although this mechanism of knowledge-based intrusions is being used to detect attacks in this project, the underlying detection model lends itself to other mechanisms such as Colored Petri Nets (CPNs).

2.2.2. Colored Petri Nets

Colored Petri Nets (CPNs) [2] can also be used in order to represent signatures of attacks on encrypted exchanges between principals. Positive aspects of petri nets are their generality, simplicity and the fact that they can be represented graphically. Complex signatures can be represented easily using colored petri nets due to their generality.

3. Scope of the Project

The Intrusion Detection Engine described here, uses knowledge based signature analysis to detect attacks on encrypted exchanges between principals. The scope of the project, underlying assumptions and the role of this prototype in SEADS is discussed below.

3.1. Role of the IDE in SEADS

The Secure Enclave Attack Detection System (SEADS) [6] is a network monitoring system. Security protocols, which constitute encrypted exchanges between principals, are excellent targets for attack by sophisticated intruders and are vulnerable to such attacks. SEADS monitors executing protocols and detects malicious and questionable activity by leveraging protocol-specific knowledge gleaned from formal method analysis and other sources. SEADS protects trusted services and their users through dynamic analysis of security protocols. It does this by analyzing ongoing secure enclave transmission activity and comparing it against an accumulated knowledge base.

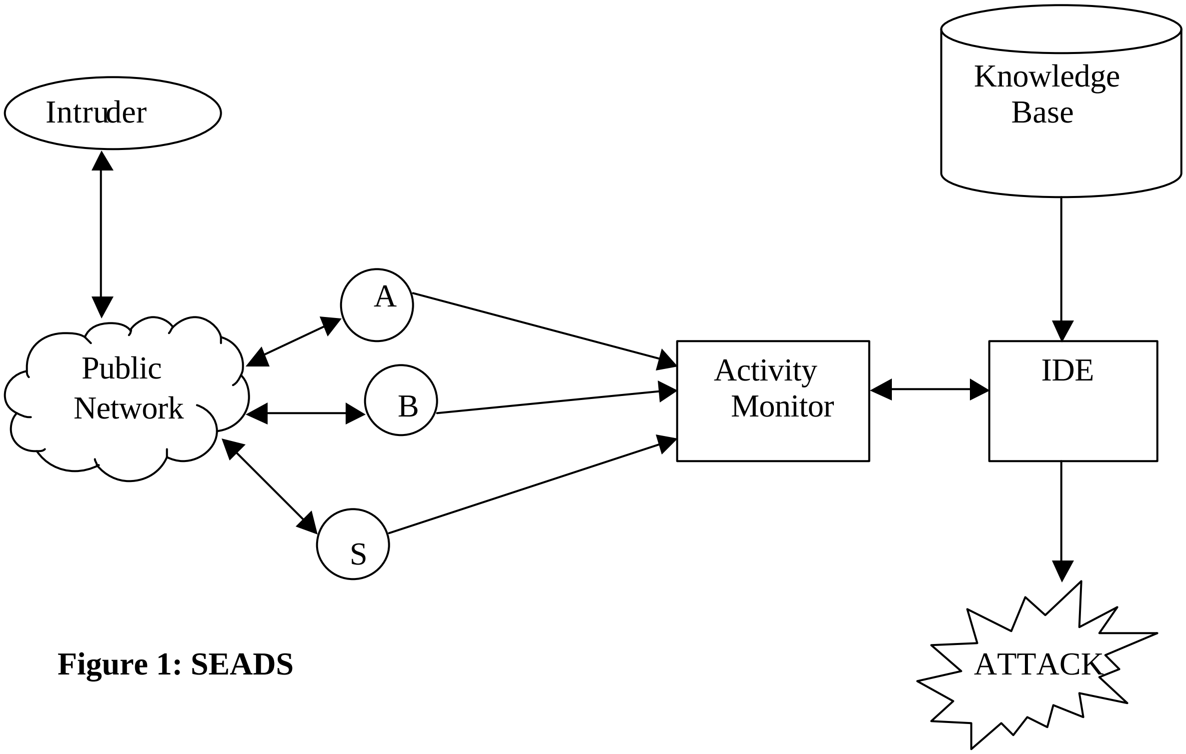


Figure 1: SEADS

The three components of SEADS are the *Activity Monitor*, the *Knowledge Base* and the *Intrusion Detection Engine*.

1) The Activity Monitor: The activity monitor is responsible for keeping track of the ongoing security protocol activity. For instance, whenever a valid principal wishes to start a secure protocol session with another valid principal, it informs the activity monitor of its intent. Say a principal (A) wishes to start a protocol session. Each time one of the principals sends or receives a message from another principal, it informs the monitor about this event, and also provides information about this event like protocol name, session number and message number. If we assume protected communication between the monitor and principals, the monitor will always have total information about all the protocol sessions being executed within the secure enclave. This information allows the IDE to detect attacks.

2) The Knowledge Base: The knowledge base component of SEADS is a database of attack signatures on the various security protocols. This database is populated based on previous research or newly discovered attacks on security protocols. The knowledge base grows dynamically as new attack signatures are added.

3) The Intrusion Detection Engine: The IDE is responsible for detecting anomalous protocol activity within the secure enclave. It achieves this by comparing the message events sent to it by the monitor against the stored attack signatures in the knowledge base. If any protocol activity is found to be an attack or suspicious, the IDE will generate an appropriate signal.

3.2. Assumptions

The following assumptions were made during the design and development of the IDE prototype.

- Communication between the Monitor and principals is reliable and secure.
- Communication between the principals themselves is not secure, i.e. a principal may send a message to another principal and tell the monitor that it did so, but the

monitor may never receive the message receipt confirmation from the other principal.

- Communication between the Monitor and the IDE is reliable and secure.

Based upon research done during the development and design of this project, the following points should be noted.

- The IDE can classify its detection in two broad categories: single session attack detection and multi-session attack detection.
- The IDE attack detection is limited to those attack signatures, which exist in the SEADS knowledge base.

3.3. Functionality

The IDE prototype provides the following functionality.

a) All attacks whose signatures exist in the knowledge base are signaled as positive attacks.

b) Any attack whose signature does not exist in the knowledge base is signaled as suspicious.

c) If any protocol session terminates abnormally, the IDE will report this activity as being suspicious.

d) Normal protocol activity will not cause any notification signals.

e) The attack reports generated by the IDE will provide full details including:

   - Principals involved in the attack session.
   - Protocol Name
   - Signature of the attack (if existing in the knowledge base)
   - Time of the attack.

f) A graphical user interface (GUI) which allows the user to generate attack reports for attacks on specific protocols, or attacks during specific times. The GUI will also allow the user to back up the active attack report file to which the IDE writes the detected attack information.

g) Along with the above functional features, the IDE will consistently interface with the other two modular components of SEADS i.e. the activity monitor and the knowledge base.

4. Detection Methodology

As mentioned in section 2.2.1 the detection mechanism of the IDE is based on the knowledge-based paradigm. The IDE detects anomalous, malicious or suspicious protocol activity occurring within the secure enclave of valid principals.

4.1. Constructing Signatures of Attacks

A very important feature of the IDE is that the detection mechanism does not rely upon knowledge of the payload of the messages exchanged between the principals during protocol sessions. This is because the IDE detects attacks based upon the signatures of attacks on security protocols. We show how the formal definition of a security protocol is translated into a signature, which can be used by the IDE to detect attacks.

Consider the *Needham and Schroeder Conventional (symmetric) Key Protocol* (NSCKP) described in [5]. This protocol constitutes three principals: A, B and the trusted third party server S. The aim of this protocol is to establish a secret key Kab that is to be used by the principals A and B to encrypt their future exchanges.

<u>Notation:</u>

- Text following the " : " symbol constitutes the actual message payload exchanged between principals.
- E(K : D1, D2, D3) means that the plain text message (D1, D2, D3) is encrypted under the key K.
- An arrow pointing right (A → B) indicates a message sent from A to B.
- An arrow pointing left (B ← A) indicates a message received by B from A.

The protocol is described as below:

1) A → S : A, B, Na
2) S → A : E(Kas: Na, B, Kab, E(Kbs : Kab, A))
3) A → B : E(Kbs : Kab, A)

4) B → A : E(Kab : Nb)
5) A → B : E(Kab : Nb – 1)

Here is a step-wise description of what the protocol achieves.

Step 1: The principal A requests from the server S, a key to be used for secure communication with B. A includes a random number generated specially for this run of the protocol. This nonce (random number) Na is used to ensure that message (2) is timely.

Step 2: S creates a key Kab (the secret key for both A and B to use in future exchanges), and creates message (2). Only A can decrypt this message successfully because it is encrypted with the key that it shares with S (Kas). In doing so A will obtain the key Kab and check that the message contains the nonce Na.

Step 3: A passes to B the encrypted message component E(Kbs: Kab, A) as message (3).

Step 4: Principal B decrypts this message to discover the key Kab and that it is to be used in communication with A. B then generates a nonce Nb, encrypts it (using the newly obtained key Kab), and sends the result to A as message (4).

Step 5: Principal A, who possesses the appropriate key Kab, decrypts it, forms Nb – 1, encrypts it and sends the result back to B as message (5). B decrypts this and checks that the message is correct.

At the end of a correct run of the protocol, both principals should be in possession of the secret key Kab newly generated by the server S.

The above description of the protocol [1] in five steps includes information about the payload data exchanged by the principals. However as previously mentioned, the IDE does not rely on this information for its detection mechanism. The above protocol description can be abstracted out into the following signature:

1) A → S
2) S ← A
3) S → A
4) A ← S
5) A → B
6) B ← A
7) B → A
8) A ← B
9) A → B
10) B ← A

Each step of the above signature can be considered to be an *event*. Thus A sending a message to S is considered as a 'send' event from A to B and similarly S receiving a message from A is a 'receive' event by S from A. An important feature of the above protocol signature is that it also includes the message receiving events. This means that principal A sending a message to principal S (as in event 1), and correspondingly principal S receiving the same message (event 2) will be represented as two distinct events in the protocol signature used by the IDE. All the events marked in bold are the message receive events.

Consider a scenario during the run of the above protocol in the secure enclave. If principal A sends a message to the server S as part of the first step of the protocol, it will inform the activity monitor of SEADS about this. Since a public network is being used for the message transfer between A and S on insecure lines, the message may get lost or may be intercepted by an intruder. In either case the server S will never receive the message sent by A. As a result S will not inform the monitor that it actually received a message from A. Thus, the sequence of events logged in the monitor will show a message sent by A to S, but not received by S, as evident by the lack of the receive notification by S to the monitor. It is prudent therefore to include the message receipt as a separate event in the protocol signature.

The above signature comprising of ten events is actually a signature of a normal run of the Needham and Schroeder Conventional Key Protocol.

Let us illustrate how the above protocol is subject to an attack, which compromises what the protocol is actually intended to achieve. This attack on the Needham and Schroeder Conventional (symmetric) Key Protocol was demonstrated by Denning and Sacco [4].

Consider message (3) of the actual protocol description given above. Although B decrypts this message and assumes legitimately that it was created by the server S, there is nothing in the message to indicate that it was actually created by S as part of the current protocol run. Thus, suppose, a previously distributed key K'ab has been compromised (say by cryptanalysis) and is known to an intruder Z. Z might have monitored the network when the corresponding run was executed and recorded message (3) consisting of E(Kbs: K'ab, A). He can now fool B into accepting the key as new by the following protocol.

(3) Z(A) → B : E(Kbs : K'ab, A)
(4) B → Z(A) : E(K'ab : Nb)
(5) Z(A) → B : E(K'ab : Nb –1)

Note: The notation Z(A) stands for *Z masquerading as A.*

Now, B believes he is following the correct protocol. Z is able to form the correct response in (5) because he knows the compromised key K'ab. He can now engage in a communication with B using the compromised key and masquerade as A.

Let us generate a signature recognizable by the IDE for the above attack on the Needham and Schroeder protocol.

1) B ← A
2) B → A
3) B ← A

The above signature comprises of only three events, two receive events and a send event. Since the malicious intruder (Z), is not part of the secure enclave, it will not co-operate with the activity monitor and hence will not inform the monitor whenever it sends or receives messages. Thus the above attack signature will consist only of events, reported by principal B (a valid principal), to the monitor.

4.2. Use of Finite State Machines

In section 4.1 we have described in detail how the attack signatures are constructed from the description of security protocols. The IDE interfaces with the activity monitor to receive events corresponding to protocol sessions executing within the enclave and compares the events with the attack signatures stored in the knowledge base. The comparison mechanism in the IDE is achieved by using Finite State Machines.

Each time the IDE receives an event from the monitor, which corresponds to the first event of a new protocol session, it looks in the knowledge base and constructs a finite state machine for each signature stored in the knowledge base for that particular protocol.

4.2.1. Signature Format in the Knowledge Base

Each signature is stored in the Knowledge Base in the following format:

```
begin XX NUM type
state1 principal (→/←) principal state2 msgNum sessNum
state1 principal (→/←) principal state2 msgNum sessNum
--
```

--

--

--

end

<u>Notation:</u>

begin/end: Indicates that the signature starts/ends at this line.

XX: The abbreviated name of the protocol.
NSCKP – Needham and Schroeder Conventional Key Protocol

NUM: A numeric value which indicates if the signature to follow is one of a correct run of the protocol or an attack.
NUM >= 0 indicates that the signature is an attack signature
NUM = -1 indicates that the signature is that of a normal run of the protocol.

type: Indicates the type of attack this signature represents. Possible values are:
S – Single session attack.
R – Replay attack.
P – Parallel session attack.
(*See section 4.3 for a discussion about the above three attack types*.)

state1: Indicates the Finite State Machine (FSM) state before the event (send or receive) takes place. (ss: start state, s1:state 1, etc.)

state2: Indicates the Finite State Machine (FSM) state after the event (send or receive) has taken place. (s1:state 1, …. fs :final state)

→/← send or receive event.

principal: May be the single letter identifiers of principals viz. A, B, S etc.

msgNum: The message sequence number in the current run of the protocol.

sessNum: Unused field that may be used to add protocol session numbers if needed.
This field will always be set to 1 in the current implementation.

Consider an actual signature in the knowledge base. This signature is of the attack described in section 4.1 on the Needham and Schroeder Conventional Key Protocol (NSCKP).

begin NSCKP 0 S
ss B ← A s1 1 1
s1 B → A s2 2 1
s2 B ← A fs 3 1
end

4.2.2. Construction of the Finite State Machine

The IDE constructs Finite State Machines for each signature stored in the knowledge base, corresponding to a protocol session that the IDE is monitoring for attacks. The state transition diagram for attack signature #0 on the NSCKP protocol (as described in section 4.2.1) is shown below.

Current State	Event	Next State	Message Number
SS	B ← A	S1	1
S1	B → A	S2	2
S2	B ← A	**FS**	3

Initially the transition machine will be in the start state (SS). As the IDE receives events from the monitor for this particular protocol session it will advance the FSM (for this signature) if the events coming in match those in the attack signature. Upon a transition to the final state (in any of the finite state machines corresponding to the attack signatures of the protocol) the IDE will signal an attack notification if the signature is an attack signature. If the finite state machine corresponding to the normal run of the protocol reaches the final state (FS) the IDE will not raise any notification alerts, since a correct run of the protocol has just concluded.

4.3. Attack Detection

The IDE uses distinct detection methodologies for protocol attacks, depending on the number of sessions over which the attack takes place. Attacks on security protocols may be over only a single session of the protocol or may utilize information gleaned over previous runs of the protocol. Thus, attacks detected by the IDE may be classified as:

1.Single session attacks
2.Multi-session attacks

4.3.1. Single Session Attacks

Single session attacks are those attacks which may or may not use information gleaned over previous sessions of the protocol like encryption keys. However they are not related temporally in any way with those sessions. The attack demonstrated on the Needham and Schroeder Conventional Key Protocol (NSCKP) in section 4.1. can be considered as a single session attack even though its success depends on a previously compromised key from another session. The 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.

4.3.2. Multi-Session attacks

Multi-session attacks are those attacks which use information gleaned over previous sessions of the protocol. However such attack sessions have to use the information within a certain time period of the reference sessions (from which the information is taken), in order to successfully subvert the protocol.

For the purposes of detecting multi-session attacks the IDE classifies all of them into one of two categories:

- Replay Attacks
- Parallel Session Attacks

4.3.2.1.Replay Attacks

Replay attacks use information extracted from a normal run of a protocol. This information is used before it expires (say within the FRESHNESS time, if the information is actually a key) within the attack session of the same protocol. The detection of such an attack on a protocol by the IDE will not only depend on the successful passage through all the states of the finite state machine for the attack signature, but also on the timing aspects relative to a normal run of the protocol.

The IDE will signal such activity as a positive attack only if it occurs within 10 seconds of a normal run of the protocol. The wait constant has been chosen to be 10 seconds for the IDE prototype, for demonstration purposes only. However the actual FRESHNESS time may vary from protocol to protocol and may also depend on the type of application being run by the principals. 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.

4.3.2.2.Parallel Session Attacks

A parallel session attack occurs when two or more protocol runs are executed concurrently and messages from one are used to form messages in another. As a simple example consider the following one-way authentication protocol [1]:

1) A → B : E(Kab : Na)
2) 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 message (1). 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.

1.1) A → Z(B) : E(Kab : Na)

2.1) Z(B) → A : E(Kab : Na)

2.2) A → Z(B) : E(Kab :Na + 1)

1.2) Z(B) → A : E(Kab : Na + 1)

Here A initiates the first protocol with message (1.1). Z now pretends to be B and starts the second run of the same protocol, with message (2.1), which is simply a replay of message (1.1). A now replies to this challenge with message (2.2). But this is the precise value A expects to receive back in the first protocol run. Z therefore provides this message as message (1.2).

The IDE is capable of detecting such attacks by checking whether the time for which the first protocol blocks at a particular step is long enough for another complete run of the protocol to complete. After the second run is complete, the first protocol resumes and completes its remaining steps.

4.3.3. Limitations to the Attack Detection Capabilities of the IDE

There are certain types of attacks on security protocols called *man-in-the middle* attacks, where a malicious intruder may intercept exchanges between valid principals of the enclave. However, the intruder modifies the message in such a manner that the structure of the message is still what the receiving party expects. As a result principal A might send a message to principal B. This message is intercepted by an intruder (say M). M now modifies the message and forwards it to B.

Within the secure enclave, A reports the sending of the message to the activity monitor and B also reports the receiving of the message from A, to the monitor. In such a manner the entire protocol exchange will look like a correct run of the protocol to the IDE, and such attacks may not be detected.

However strictly speaking, this limitation is actually a fundamental flaw of the protocol, which allows messages to be altered in such a way as to maintain the correct structure of the expected message. The receiving principal will therefore not have any reason to believe that the message came from someone other than a valid principal belonging to the enclave.

5. Design of the Intrusion Detection Engine

This section will provide an insight into the design that went towards implementing the Intrusion Detection Engine. A justification of the major design decisions is also given in the section. The design of the IDE uses the object-oriented paradigm. The entire problem was broken down into smaller components, and appropriate classes were developed for entities, where necessary.

5.1. Architectural Design

Here are some of the issues, which had to be taken into account in the design phase of this project.

- Handling multiple concurrent sessions of different protocols executing within the enclave and keeping track of each.
- Detecting attacks spanning over different active protocol sessions (Parallel session and replay attacks).
- Ensuring detailed reporting of detected attacks and/or suspicious activities.
- Interfacing with the Monitor and the Knowledge Base
- Maintaining a consistent and reliable attack detection capability.

5.1.1 Design Decisions

The IDE is multi-threaded with a single thread to serve as the thread dispatcher. Every time the IDE receives an event from the activity monitor, which corresponds to the first event of a new protocol session, the IDE will spawn a thread to monitor all the FSMs for that particular protocol.

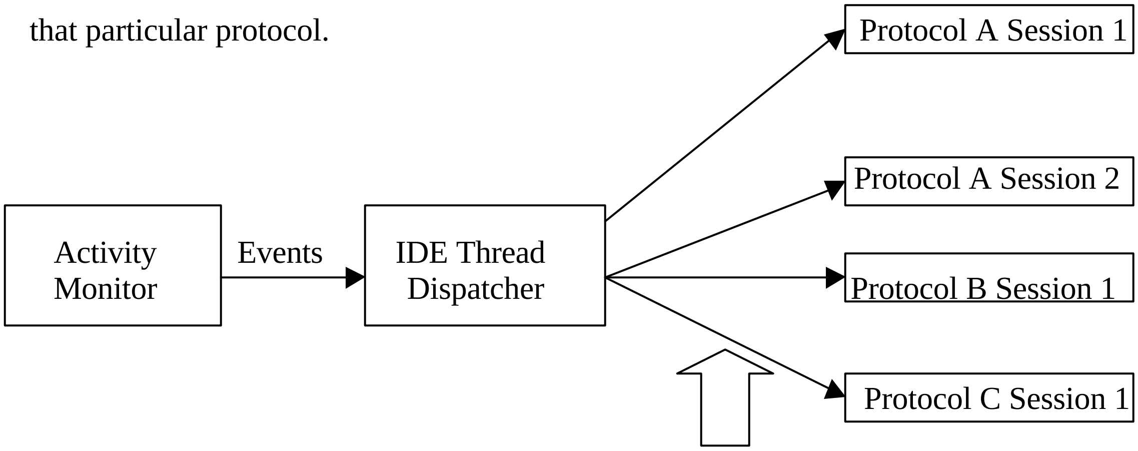


Figure 2:Thread Design Channeling the events to the relevant thread

Any further events coming in from the activity monitor belonging to a protocol session which is already being monitored by an IDE thread, are channeled to the relevant thread. To keep track of all the threads existing within the system, a *ThreadList* class was employed, which would hold the following information for each thread:

- Protocol Name
- Session Number
- Principals involved
- Thread identifier
- Signal to which the thread listens, whenever the IDE receives an event meant for it.

5.1.1.1.Functionality of Threads

Given below are the functions that each thread performs for the IDE.

- Advance the FSM for each signature (normal or attack) for that particular protocol, if an event match occurs.
- Raise alerts upon detection of an attack or conclusion of a normal protocol session.
- Write the detailed attack information to a text file.
- A thread will die upon detection of an attack/normal session or suspicious activity. However if a particular protocol 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 *ThreadList* will be removed.

The reason threads are chosen as an option for the IDE is:

- The upper limit on the number of concurrent threads spawned by a process is limited only by the virtual memory on the system.
- 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.

5.2. Design Flowchart

The overall design of the IDE can be represented by means of flow chart for greater clarity. (See figure 3.)

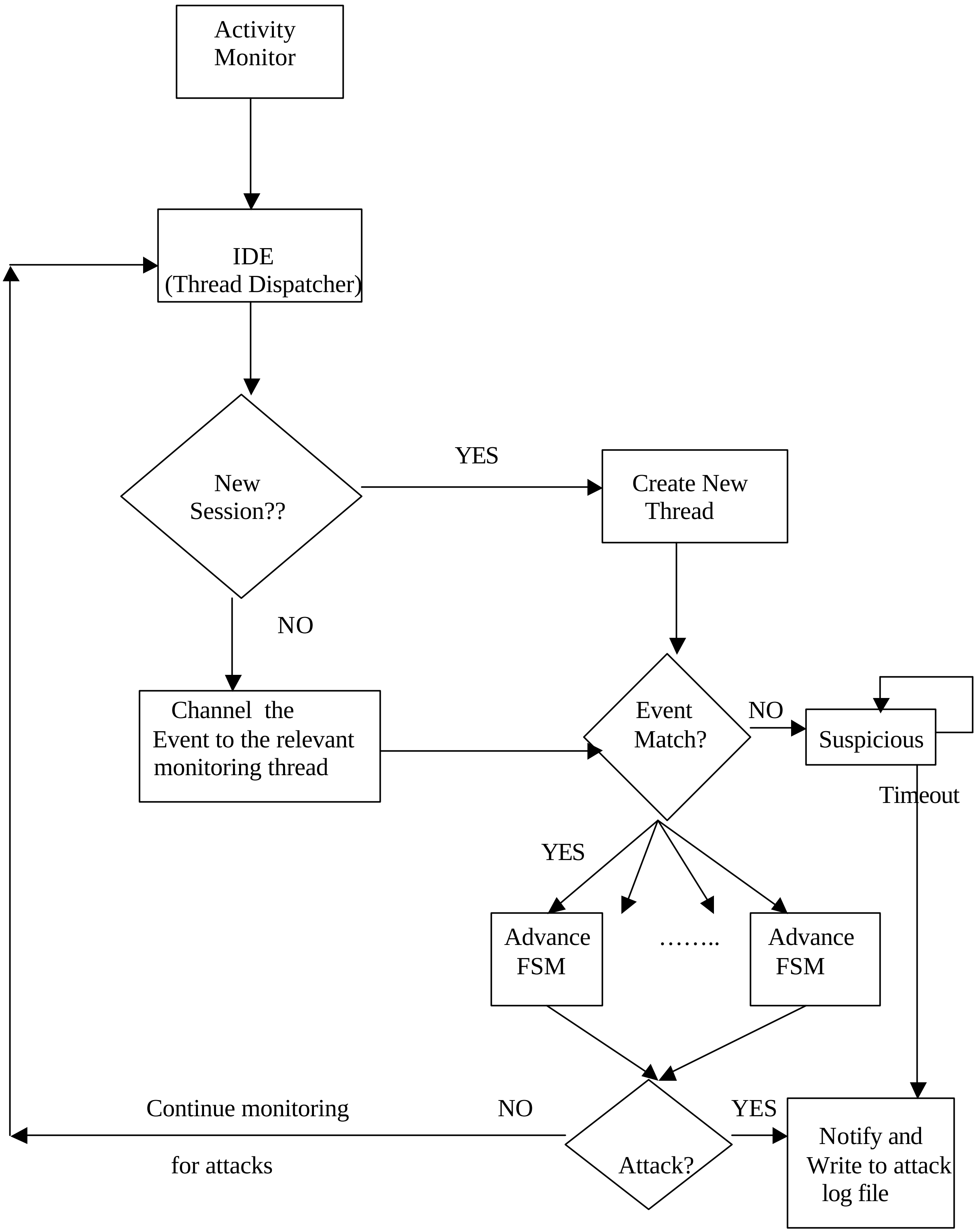


Figure 3: Design Flowchart

6. The Graphical User Interface

In this project, a graphical user interface (GUI) was implemented for an overall view of the attacks / suspicious activities detected within the enclave. The GUI provides the following capabilities.

6.1. Reporting Attacks to the user

The user can specify the time duration and the protocol name to obtain a detailed report of all the attacks (on the specific protocol) that took place during that period.

The report includes:

- The name of the protocol subject to attack.
- The principals involved in that session.
- The attack signature from the Knowledge base corresponding to that attack
- The time that the attack took place.
- The type of that attack.

6.2. Backup of the Active Attack Log File.

The GUI also allows the user to back up the active attack report file to another file. The GUI will still read from both files to create the attack reports, but the IDE threads will only be writing to the newly empty active file.

7. Testing

Upon completion of each significant milestone, the IDE was tested to ensure that the existing product as well as the new additions functioned correctly. Test cases were developed in order to ensure that the IDE does indeed provide all the functionality that is mentioned in section 3.3.

Accordingly we will mention each individual functional claim and provide the test results to prove that it is satisfied.

a) Detection of Positive Attacks: The environment being monitored by the IDE was systematically subjected to attacks of each of the three types viz. Single session, Replay and Parallel Session attacks. The IDE was able to detect all three types of attacks.

b) Detection of suspicious activity: Event sequences not corresponding to any attacks or normal protocol runs were simulated for protocols. The IDE was correctly able to report such activity as unrecognizable suspicious activity.

c) Detection of abnormal termination: A protocol session was simulated in which there is abnormal termination before the current run has reached its completion. In such a case, the IDE thread monitoring this session, times out after the TIMEOUT period and reports abnormal termination of the protocol.

d) Detailed Reporting Capability: In each of the above three cases, the IDE will generate a detailed report about the attack and add this information to the attack log file.

The attack reports generated by the IDE will provide full details including

- Principals involved in the attack session.

- Protocol Name
- Signature of the attack (if existing in the knowledge base)
- Time of the attack.
- Type of the attack.

e) Graphical User Interface: The GUI was tested to check whether a user could get a comprehensive report on the attacks specified by him, over the specified time duration.

f) Consistent Interface with the Activity Monitor: During its normal functioning, the IDE seamlessly interfaced with the Monitor and there were no losses of events between these two modules, or any loss of functionality of one due to the other.

8. Conclusion

In this project I have designed and implemented a Knowledge Based Intrusion Detection Engine to detect attacks on security protocols executing within a secure enclave. This project will provide the necessary extra level of protection for encrypted exchanges, which was mentioned in section 1.2.

Extensive research on the characteristics of security protocols enabled this detection methodology to achieve its desired functionality. Extracting the description of security protocols into sequences of events, allows the IDE to detect attacks on those protocols.

The IDE will detect any attacks or suspicious activity on security protocols executed by valid principals operating within a secure enclave. The detection of the IDE compares protocol activity gathered by the Activity Monitor against the attack signatures stored in the Knowledge base.

A Graphical User Interface (GUI) was also developed in order to facilitate an overall report of attacks that have been detected by the IDE, along with their occurrence times.

9. Future Work

In this project the Intrusion Detection mechanism uses the Knowledge Based paradigm and incorporates state transition analysis to represent attacks on protocols. A further direction that this research could take is to use Colored Petri Nets (CPNs) mentioned in section 2.2.2. to detect attacks on the protocols.

Consider an example [2] of the functionality provided by CPNs for intrusion detection. If our knowledge base contains information, that a person attempting to login 5 times unsuccessfully within the span of one minute, is a malicious intruder, then we have the following CPN.

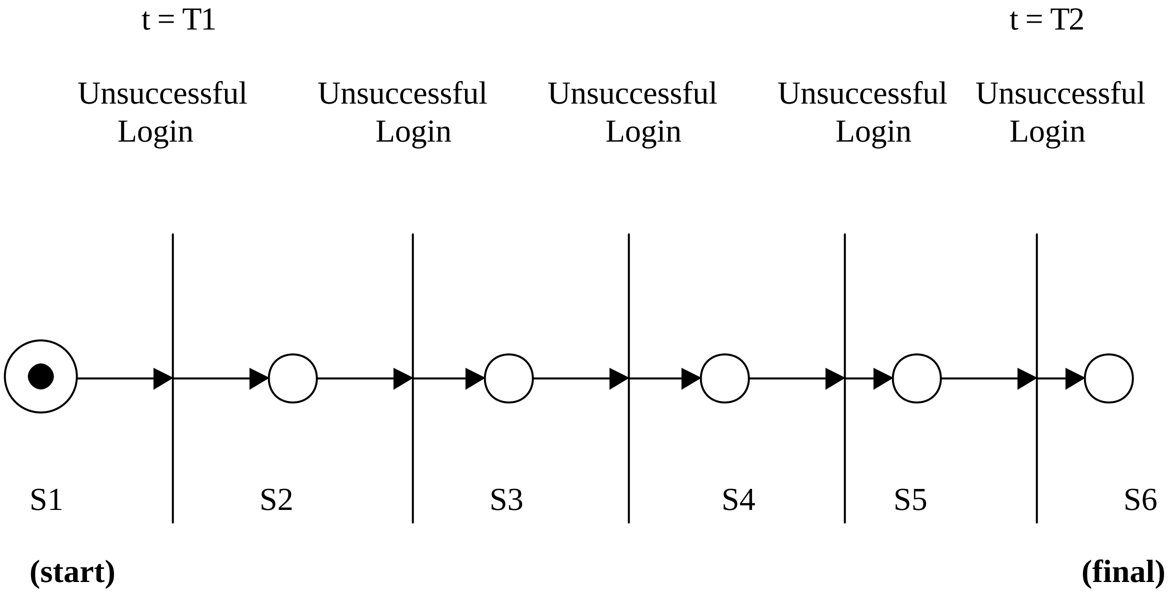


Figure 4. CPN

Figure 4. shows a CPN, which will issue an alarm when the number of unsuccessful logins within one minute exceeds five. Each vertical bar represents the transitions. A transition, from say state S1 to state S2 can only occur if there is a token in state S1 and there has been an unsuccessful login attempt. The time of the first unsuccessful login attempt is stored in the token variable t. The final transit ion from state S5 to state S6 can only occur if there is a token in S5, an unsuccessful login attempt and the time difference between this attempt and the first unsuccessful login attempt is less than a minute.

Behavior based intrusion detection is also another possible way to detect intrusions in secure enclaves. Neural networks could be used in order to learn legitimate protocol behavior and use this as a leverage to detect any anomalous or abnormal activity in the secure enclave.

10. Appendix

A. Resources used during the project

Project Development Environment: This project was implemented using the Microsoft Visual C++ development environment. All the code was written in C++.

For the Graphical User Interface, the Microsoft Foundation Class library was used to develop the graphical windows and other components.

In order to manage the multi-threading aspects of the implementation, the Win32 API was also used.

Lessons from this project

Over the course of this Master's project I was able to enhance and hone my skills in a number of aspects of Computer science. I learnt a great deal about Network Security and the execution of security protocols. In addition to the implementation aspects of this project, a major issue was handling the interfaces with the other modules of SEADS, which was a good experience. Due to the nature of this project I learnt a lot about multi-threaded Windows programming and developing graphical user interfaces.

B. Class Diagrams

The Class Diagrams for classes which provide the core functionality of the back end of the IDE, along with a brief description of their attributes are given below. Each class diagram has a header that is the name of the class. All the attributes and the methods are given below the header.

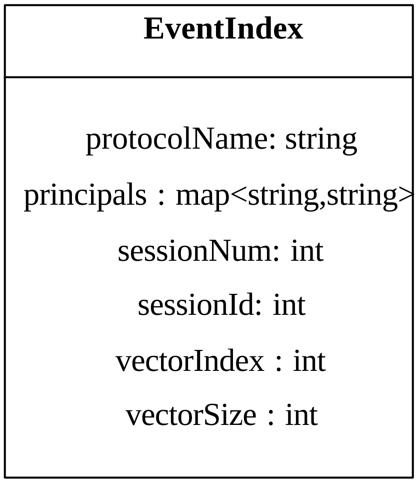
1)

Event
Attributes p1: string type: string p2:string msgSeqNum : int timestamp: time_t

Objects of the event class are used to store information about a single event such as
A → B or B ←A

- p1 is a string, which represents the first principal of an event.
- p2 is a string, which represents the second principal of an event.
- type is a string which holds the type of the event i.e. “send” or “receive”
- msgSeqNum is an integer, which holds the message sequence number in the protocol.
- timestamp is an object of the time_t class used to hold the time at which the event occurs.

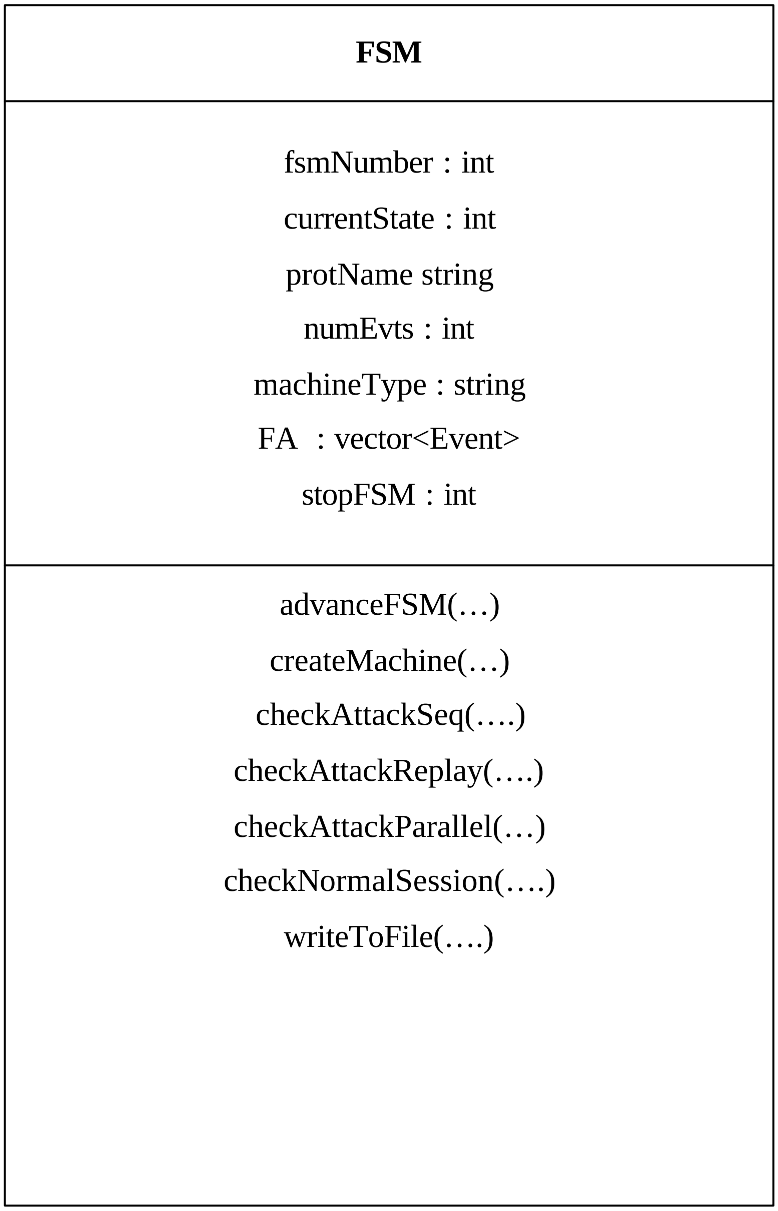
2)



The objects of the EventIndex class are used to reference individual events in the data structure of events maintained by the Activity Monitor.

- protocolName: string value which holds the protocol name for that particular event.
- principals: Standard template Library (STL) map structure which maps actual principal names to single letter names like A, B, S etc. This variable indicates the principals involved in the protocol.
- sessionNum: an integer value, which stores the session number (sequential) for each protocol involving the same principals.
- sessionId: A number generated by the principals when they begin a protocol session.
- vectorIndex: Index of the event in the data structure which is a STL vector.
- vectorSize: Length of the protocol session (in terms of number of events) as stored in the data structure.

3)



FSM

fsmNumber : int
currentState : int
protName string
numEvts : int
machineType : string
FA : vector<Event>
stopFSM : int

advanceFSM(…)
createMachine(…)
checkAttackSeq(….)
checkAttackReplay(….)
checkAttackParallel(…)
checkNormalSession(….)
writeToFile(….)

Objects of the FSM class are used to hold all information about a single finite state machine for a particular signature existing in the knowledge base. This class is the software implementation of the finite state machine.

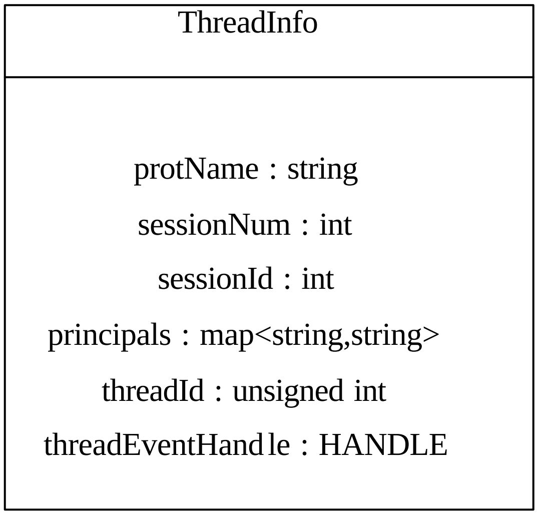
- fsmNumber: integer value, which indicates which signature in the Knowledge base, the FSM corresponds to.
- currentState: integer, which holds the current state of the FSM.

- protName: string value, which indicates which protocol this FSM has been created for.
- numEvts: number of events existing in the knowledge base signature corresponding to this FSM.
- machineType: string, which indicates if this FSM is one monitoring, attacks on a normal protocol signature or an attack signature.
- FA: a STL vector structure, which holds all the events corresponding to the signature for this FSM.
- stopFSM: a flag indicating if the FSM is still active or terminated.

Methods:

- advanceFSM( ): This method will advance the state of the finite state machine upon occurrence of an event match.
- createMachine( ): This method reads the knowledge base of signatures and populates the FSM object with relevant information based on reading a particular signature.
- checkAttackSeq( ) : This method checks whether the FSM has advanced to its final state for a single session attack to be signaled.
- checkAttackReplay( ) : This method checks whether the FSM has advanced to its final state for a replay attack to be signaled.
- checkAttackParallel( ) : This method checks whether the FSM has advanced to its final state for a parallel session attack to be signaled.
- checkNormalSessionl( ) : This method checks whether the FSM has advanced to its final state for a normal run of the protocol to have concluded.
- writeToFile( ): This method will write the information about the detected attack or suspicious activity to the attack log file.

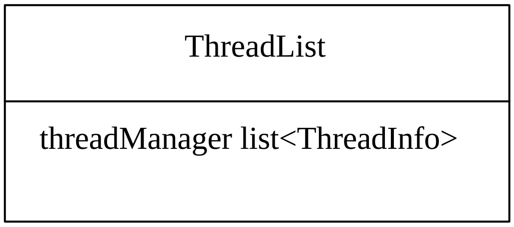
4)



Objects of the ThreadInfo class hold information about each thread monitoring individual protocol sessions. Each protocol session is uniquely identified by the protocolName, session number and the principals involved in the protocol.

- protName: string variable, which is used to store the protocol name.
- sessionNum: an integer value, which stores the session number (sequential) for each protocol involving the same principals.
- sessionId: an integer value which is a random number generated by the principals.
- threadId: an unsigned integer value (corresponding to DWORD in Win32) used to store the thread identifier.
- threadEventHandle: a Win32 HANDLE to the signal event signaled by the dispatcher IDE thread, when it receives an event belonging to a protocol session which is being monitored by this thread.

5)



An object of this class is used to store the global information about all the threads spawned by the IDE, which are currently active.

- threadManager: This is an STL list data structure comprising of objects of the ThreadInfo class.

C. The IDE Source Code

The code that went into the project is attached in the following pages. The code is organized into the following files:

IDE Backend

1.Seads_Util.h: This header file contains the class declarations and global function prototypes, which are used by the IDE.

2.Seads_Util.cpp: This is the implementation file containing the function definitions of the various class member functions and global functions.

3.Seads_General.h: This header file contains classes shared between the Activity monitor implementation and the IDE.

4.Seads_Monitor.cpp: This implementation file contains the code for the interface between the Activity Monitor and the IDE. The Activity Monitor actually creates an IDE thread, which runs continuously. Only relevant code from this file has been attached.

IDE Graphical User Interface

1.Dialogs.h: This is the header file, which contains the declarations for the Attack Report Dialog Box class.

2.Dialogs.cpp: This file contains the implementation for the methods declared in *"Dialogs.h"*.

3.CIDEDoc.h: This header file contains the declarations for the CIDEDoc class (a Document class), which is part of the Document/View, architecture of the Microsoft Foundation Class library (MFC) used to develop the GUI.

4.CIDEDoc.cpp: This file contains the CIDEDoc class method definitions.

5.CIDEView.h: This header file contains the declarations for the CIDEView class (a View class) - another part of the Document/View architecture of MFC.

6.CIDEView.cpp: This file contains the CIDEView class method definitions.

Some of the other code, which is primarily generated by the development environment, has not been attached.

```
///////////////
// SEADS_Util.h
///////////////


#ifndef _SEADS_UTILITY_H
#define _SEADS_UTILITY_H

#include <string>             // For the string STL container
#include <vector>             // For the vector STL container
#include <map>                // For the STL maps
#include <list>               //

#include <process.h>

#include <fstream>
#include <ctime>

#include "SEADS_General.h"

#define KBASE "SEADS_Signature_File.txt"
#define OUTFILE "attackReport.txt"
#define OUTFILE_GUI "report.txt"

using namespace std;


// Class to keep track of each session monitoring thread
// Holds all the relevant information for a single thread
class ThreadInfo
{
public:

    string protName;               // protocol name
    int sessionNumber;             // session number (sequential)
    int sessionId;                    // session nonce    (random generated)
    nameMap principals;               // principals involved
                                      // protName, sessionNumber and principals uniquely
identifies a session
    DWORD threadId;                // unique thread identifier
    HANDLE threadEventHandle;      // handle to the event which the dispatcher thread will
signal.
};


// Class to keep track of all session threads created.
class ThreadList
{
   public:
     list<ThreadInfo> threadManager;     // list of individual ThreadInfo objects.
                                         // corresponding to the number of alive sessions.

};


class GlobalVariables
{
public:
    GlobalVariables();
    ~GlobalVariables();

    // Globals shared between dispatcher thread and FSM threads
    CRITICAL_SECTION outfile_cs;  // critical section for the output file
    CRITICAL_SECTION tList_cs;    // critical section to access the ThreadList
    CRITICAL_SECTION cout_cs;     // critical section for console output
    ThreadList tList;
};
```

```
// Class for shared objects between the monitor and the IDE.
class ThreadParameters
{
public:
    GlobalVariables gv;
    HANDLE evtHandle;
    Monitor *mPtr;
};



// FSM class which holds a SINGLE attack/ normal signature.
class FSM
{

public:

  FSM();
  int advanceFSM(ThreadParameters *,Event,EventIndex );
  void createMachine(int, vector<string>, int);
  int checkAttackSeq(ThreadParameters *,EventIndex);
  int checkAttackReplay(ThreadParameters *,EventIndex );
  int checkAttackParallel(ThreadParameters *, EventIndex);
  int checkNormalSession(ThreadParameters *);
  int writeToFile(ThreadParameters *, EventIndex, string);
  int stopFSM;
  string detectType; // if attack detected by FSM..what type?

private:

  int fsmNumber;

  int currentState;
  string protName;
  int numEvts;
  string machineType;   // attack or normal
  vector<Event> FA;
  string attackType;    // sequential or replay or parallel session. (S/R/P)
  int blockTime;        // time for which the session should block for parallel session
attacks
  int blockEvt;         // event at which the session should block.

};

// Thread Procedure
unsigned __stdcall SessionAttackFunc ( LPVOID holder );


// Utility functions
void getThreadInfo(ThreadParameters *, ThreadInfo *, DWORD);
HANDLE getEventHandle (ThreadParameters *,EventIndex *ei);
void evtToString(Event, char *);
int numSignatures(const char *, vector<string> *, int *);
int eventCompare(Event, Event);

int writeSuspicious(ThreadParameters *,EventIndex ,string,string);
void removeThreadEntry(DWORD, ThreadParameters *);
int checkTime(ThreadParameters *,EventIndex );
int checkTimeParallel(ThreadParameters *,EventIndex, int, int );

#endif
```

```
/***************************************
 SEADS_Util.cpp
 Sachin Goregaoker
 Last Modified: 7/5/2001
 Intrusion Detection Engine
****************************************/


/*****************************************************************
 This File contains the utility function definitions as well as
 the various class member function definitions for the IDE.
 The dispatcher thread method is contained in this file.
******************************************************************/



#include <string>        // For the string type
#include <iostream>
#include <fstream>       // File reading and writing
#include "SEADS_Util.h"  // Header file with the class definitions


/*****************************************************************
 This function compares two Event class objects and determines if
 they are the same.
 Return value: 1 if there is an event match.
               0, otherwise
 *****************************************************************/

int eventCompare(Event A, Event B)
{
  if( A.p1==B.p1 && A.type==B.type && A.p2==B.p2 )
      return 1;
  else return 0;
}

// Constructor for the class GlobalVariables
// Contains the initializations for the critical sections

GlobalVariables::GlobalVariables()
{
    InitializeCriticalSection(&outfile_cs);
    InitializeCriticalSection(&tList_cs);
    InitializeCriticalSection(&cout_cs);
}

// Destructor for the class GlobalVariables.
// Deletions of the critical sections are contained in the destructor.

GlobalVariables::~GlobalVariables()
{
    DeleteCriticalSection(&outfile_cs);
    DeleteCriticalSection(&tList_cs);
    DeleteCriticalSection(&cout_cs);

}

// Constructor for the class FSM.

FSM::FSM()
{
  stopFSM=0;         // Indicating the status of the particular FSM
  numEvts=0;         // Number of events in the FSM
  currentState=0;    // Current State of the FSM
  blockTime = 0;     // blocking time for parallel session attacks.
}


/*********************************************************
```

```
 This function reads from the vector which contains the
 signatures read in from the KBASE. Then it will create
 the FSM for each signature belonging to the protocol
 being monitored for attacks.

 This function will populate the FA vector with individual
 events.
 *******************************************************/

void FSM::createMachine(int id, vector<string> buffer, int bufLength)
{

    int i=0,j=0;
    int fsmCreated=0;
    char *temp=new char[100];
    char *charid=new char[100];
    char *token;
    char *delimiter=" ";


    while(i<bufLength && !fsmCreated)
    {
        strcpy(temp,buffer[i++].c_str());
        token=strtok(temp,delimiter);


        if(strcmp(token,"begin")==0)
         {
             protName=strtok(NULL,delimiter);
             token=strtok(NULL,delimiter);


             if(strcmp(token,itoa(id,charid,10))==0)
             {

                  if(id<0)
                      machineType="normal";
                  else
                  {
                      machineType = "attack";
                    attackType = strtok(NULL, delimiter);
                  }

                  fsmNumber=id;

                  while(1)
                  {

                    Event evt;
                  strcpy(temp,buffer[i++].c_str());

                    token=strtok(temp,delimiter);

                    if(strcmp(token,"end")!=0)
                    {

                       token=strtok(NULL,delimiter);

                       evt.p1=token;
                      token=strtok(NULL, delimiter);

                      if(strcmp(token,"->")==0)
                           evt.type="send";
                       else
                            evt.type="recv";

                      token=strtok(NULL,delimiter);

                       evt.p2=token;
                       token = strtok(NULL,delimiter);
                      evt.msgSeqNum=atoi(strtok(NULL,delimiter));
```

```
                        FA.push_back(evt);
                        // get the session number
                        strtok(NULL,delimiter);
                        // get the block time if parallel session attack


                        char *s = 0;
                        s = strtok(NULL,delimiter);
                     if(s)
                        {
                           blockEvt = numEvts;
                              blockTime = atoi(s);
                        }


                        numEvts++;

                        j++;
                    }
                    else
                    {

                       fsmCreated=1;
                       break;
                    }
                  }


              }

         }


     }



}


/*************************************************************
 This function will advance the state of the FSM, and if there
 happens to be an attack, it will display the appropriate
 attack notification.

  Returns 1 if an attack protocol session has terminated
  Returns 2 if an normal protocol session has terminated
  Returns 0 otherwise.
**************************************************************/

int FSM::advanceFSM(ThreadParameters *tp,Event evt,EventIndex ei)
{

  if(!stopFSM)
  {
      if(eventCompare(evt ,FA[currentState]))
      {
               EnterCriticalSection(&tp->gv.cout_cs);
               cout<< " An event match has occurred in FSM #" << fsmNumber << endl;
               LeaveCriticalSection(&tp->gv.cout_cs);
               currentState++;

               // check if the fsm is a fsm monitoring replay attacks
               if(attackType.compare("R") == 0)
               {
                    int check = checkAttackReplay(tp,ei);
                    if(check)
                    return 1;
                    else return 0;
               }
               // single session attacks
               else if(attackType.compare("S") == 0)
               {
                   int check = checkAttackSeq(tp,ei);
                   if(check)
                   return 1;
                   else
                       return 0;
               }

               // parallel session attack
               else if(attackType.compare("P") == 0)
               {
                      int check = checkAttackParallel(tp,ei);
                      if(check)
                          return 1;
                      else
                          return 0;

               }
               else
               {
                      if(checkNormalSession(tp))
                          return 2;
                      else return 0;
               }

       }
       else
       {
            stopFSM = 1;
           return 0;
       }
  }
  else
  {
    //stopFSM=1;
    return 0;
  }


}

/*******************************************************
 This function will determine if the FSM has advanced
 sufficiently such that the attack in progress exactly
 matches one from the KBASE which is a single session
 attack.
 Returns 1 if an attack activity has occurred.
 Returns 0 otherwise.
 ******************************************************/

int FSM::checkAttackSeq(ThreadParameters *tp, EventIndex ei)
{
    if(currentState==numEvts && strcmp(machineType.c_str(),"attack")==0)
     {
         EnterCriticalSection(&tp->gv.cout_cs);
         cout << endl << "*********************************" << endl
                        << "******SINGLE SESSION ATTACK******" << endl
                        << "*********************************" << endl
                        << endl;

         writeToFile(tp,ei,"S");


         cout<< "A single session attack on the " << protName
             << " protocol [FSM #" << fsmNumber << "] has been detected!!" << endl <<
endl;
        LeaveCriticalSection(&tp->gv.cout_cs);
         stopFSM = 1;
         detectType = "attack";
         return 1;


    }

    else return 0;
}

/*******************************************************
 This function will determine if the FSM has advanced
 sufficiently such that the attack in progress exactly
 matches one from the KBASE which is a multi-session
 replay attack.
 Returns 1 if an attack or suspicious activity has occurred.
 Returns 0 otherwise.
 *******************************************************/


int FSM::checkAttackReplay(ThreadParameters *tp,EventIndex ei)
{
    if(currentState==numEvts && strcmp(machineType.c_str(),"attack")==0)
    {

         if(checkTime(tp,ei))
         {
             EnterCriticalSection(&tp->gv.cout_cs);
             cout << endl << "*********************************" << endl
                            << "*********REPLAY ATTACK***********" << endl
                            << "*********************************" << endl
                            << endl;

             writeToFile(tp,ei,"R");


             cout << "A REPLAY attack on the " << protName
                  << " protocol [FSM #" << fsmNumber << "] has been detected!!" << endl
<< endl;
            LeaveCriticalSection(&tp->gv.cout_cs);
             stopFSM = 1;
                  detectType = "attack";
                  return 1;
         }
         else
         {
             EnterCriticalSection(&tp->gv.cout_cs);
            cout << endl << "*********************************" << endl
                          << "*******SUSPICIOUS ACTIVITY*******" << endl
                            << "*********************************" << endl;

             writeToFile(tp,ei,"SS");
             cout << "Suspicious behaviour was detected on the "<< protName
                   << " protocol!! "<< endl << endl;
             LeaveCriticalSection(&tp->gv.cout_cs);
                 stopFSM = 1;
                  detectType = "attack";
                  return 1;
         }



    }

    else return 0;


}

/*******************************************************
 This function will determine if the FSM has advanced
 sufficiently such that the attack in progress exactly
 matches one from the KBASE which is a Parallel-session
 attack.
 Returns 1 if an attack or suspicious activity has occurred.
 Returns 0 otherwise.
 ******************************************************/


int FSM::checkAttackParallel(ThreadParameters *tp, EventIndex ei)
{
    if(currentState==numEvts && strcmp(machineType.c_str(),"attack")==0)
     {
         if(blockTime == 0)
         {
             EnterCriticalSection(&tp->gv.cout_cs);
             cout << "******************************************************" << endl
                  << "There is a possibility that a parallel session attack" << endl
                   << " is under way!!" << endl
                   << "******************************************************" << endl <<
endl;
            LeaveCriticalSection(&tp->gv.cout_cs);
             stopFSM = 1;
             detectType = "attackPP";
             return 1;
         }
         else
         {
             int check = checkTimeParallel(tp,ei,blockTime,blockEvt);
             if(check)
             {
                  EnterCriticalSection(&tp->gv.cout_cs);
                  cout << endl << "********************************" << endl
                                 << "****PARALLEL SESSION ATTACK******" << endl
                                 << "*********************************" << endl
                                 << endl;

                  writeToFile(tp,ei,"P");


                  cout << "A PARALLEL SESSION attack on the " << protName
                        << " protocol [FSM #" << fsmNumber << "] has been detected!!" <<
endl << endl;
                  LeaveCriticalSection(&tp->gv.cout_cs);
                  stopFSM = 1;
                  detectType = "attackP";
                       return 1;

             }
             else
             {
                  EnterCriticalSection(&tp->gv.cout_cs);
                  cout << endl << "********************************" << endl
                                 << "*******SUSPICIOUS ACTIVITY*******" << endl
                                 << "*********************************" << endl;

             writeToFile(tp,ei,"SS");
                  cout << "Suspicious behaviour was detected on the "<< protName
                        << " protocol!! "<< endl << endl;
                  LeaveCriticalSection(&tp->gv.cout_cs);
                  detectType = "attack";
                  stopFSM = 1;
                  return 1;
             }
         }
     }
     else
         return 0;

}


/*******************************************************
```

```
 This function will determine if the FSM has advanced
 sufficiently such that the session in progress exactly
 matches a normal session from the KBASE.
 Returns 1 if an normal session has occurred.
 Returns 0 otherwise.
 *******************************************************/
int FSM::checkNormalSession(ThreadParameters *tp)
{
    if(currentState == numEvts && strcmp(machineType.c_str(),"normal") == 0)
    {
         EnterCriticalSection(&tp->gv.cout_cs);
         cout << "*****************************************************************" <<
endl;
         cout << "A normal session of the " << protName << " protocol has concluded"
              << endl;
         cout << "*****************************************************************" <<
endl << endl;
         LeaveCriticalSection(&tp->gv.cout_cs);
         stopFSM = 1;
         detectType = "normal";
       return 1;
    }
    else
        return 0;

}

/*******************************************************
 This function is used for detecting parallel session
 attacks to check if a previously concluded normal session
 has concluded within the time frame that an attack
 session blocks.

 Returns 1 if the time elapsed between the two sessions
 [reference session and attack session] allows for the
 possiblity of a parallel session attack.
 Returns 0 otherwise.
 *******************************************************/



int checkTimeParallel(ThreadParameters *tp, EventIndex ei,int blockTime,int blockEvt)
{
    Event evtRef1,evtRef2,evtPar1,evtPar2;  // Event object holder variables.
    EventIndex tempIndex;
    ei.vectorIndex = blockEvt - 1;
    evtRef1 = tp->mPtr->queryMonitor(&ei);
    ei.vectorIndex = blockEvt;

    evtRef2 = tp->mPtr->queryMonitor(&ei);
    int numSessions;
    vector<PrincipalsSessions> ps;
    ps=tp->mPtr->querySessions(ei.protocolName);
    int size = ps.size();
    for (int i=0; i<size; i++)
    {
       if(ps[i].principals == ei.principals)
        {
           numSessions = ps[i].numSessions;
            break;
        }
    }

    int nextSession = ei.sessionNum + 1;
    for(int j=nextSession; j < numSessions; j++)
    {
      ei.vectorIndex = 0;
      ei.sessionNum = j;
      tp->mPtr->queryMonitor(&ei);
      ei.vectorIndex = ei.vectorSize-1;
      Sleep(500);
```

```
       evtPar2 = tp->mPtr->queryMonitor(&ei);
            // Check if the second interleaved session is a parallel attack session.
       if(evtPar2.type.compare("attackParallelSessionOver") == 0)
       {

           ei.vectorIndex = 0;
         evtPar1 = tp->mPtr->queryMonitor(&ei);

           if( (difftime(evtPar2.timeStamp,evtPar1.timeStamp) <= (float) blockTime)
              && (difftime(evtPar1.timeStamp,evtRef1.timeStamp) > 0)
                && (difftime(evtRef2.timeStamp,evtPar2.timeStamp) > 0))
           {
                             return 1;
           }
       }
     }

     return 0;
}

/*******************************************************
 This function is used for detecting Replay attacks
 to check if a previously concluded normal session
 has concluded within the FRESHNESS time for a Replay
 attack to possibly take place.

 Returns 1 if the time elapsed between the two sessions
 [reference session and attack session] is less than
 Freshness time
 Returns 0 otherwise.
 *******************************************************/



int checkTime(ThreadParameters *tp,EventIndex ei)
{

     const double FRESHTIME = 10;
    EventIndex holder;
     holder.principals = ei.principals;
     holder.protocolName = ei.protocolName;
     Event prevEvent,currEvent;
     currEvent = tp->mPtr->queryMonitor(&ei);

     int numSessions =ei.sessionNum;
     for(int j=0; j < numSessions ; j++)
     {
       holder.sessionNum = j;
        holder.vectorIndex = 0;
        tp->mPtr->queryMonitor(&holder);
        holder.vectorIndex = (holder.vectorSize - 1);
        prevEvent = tp->mPtr->queryMonitor(&holder);
             /* cout << " The two time stamps are :" << endl ;
              cout << " New Event: " << currEvent.timeStamp << endl;
               cout << " Old Event: " << prevEvent.timeStamp << endl;*/
               cout << "Time waited between sessions is: "
                     << difftime(currEvent.timeStamp,prevEvent.timeStamp) << endl;
        if(prevEvent.type.compare("normalSessionOver") == 0)
       {
          if(difftime(currEvent.timeStamp,prevEvent.timeStamp) <= FRESHTIME )
            {
              return 1;
               //break;
            }

        }

        else continue;


     }
    return 0;
}

/***************************************************************
 This function will enable the reporting of attack information
 by means of a text file. This file will have information like:
 Protocol name, type of attack, time of attack,parties involved.
 ***************************************************************/

int FSM::writeToFile(ThreadParameters *tp,EventIndex ei,string type)
{
     ////////////////////////////////////////
   char *lockFile = "lock.txt";
   HANDLE fHandle;
   while(1)
   {
        fHandle = (HANDLE) CreateFile(lockFile, GENERIC_READ, 0, NULL,
                                 CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);


        if(fHandle == INVALID_HANDLE_VALUE)
        {
           EnterCriticalSection(&tp->gv.cout_cs);
            cout << "File already locked" << endl;
            LeaveCriticalSection(&tp->gv.cout_cs);
            //Sleep(5000);
            CloseHandle(fHandle);

        }
        else
            break;
   }



     ////////////////////////////////////////

     //ofstream outReportFile(OUTFILE, ios::app);
     ofstream outGuiFile(OUTFILE_GUI,ios::app);
     if(!outGuiFile)
     {
         cerr << "File could not be opened for writing!" << endl;

         CloseHandle(fHandle);
        DeleteFile(lockFile);
        return 0;
     }

     /*if( !outReportFile )
     {
         cerr << "File could not be opened for writing!" << endl;

         CloseHandle(fHandle);
         DeleteFile(lockFile);
        return 0;
     }*/
     outGuiFile << "Attack" << "\n";
     outGuiFile << protName << "\n";

     time_t ltime;
     time( &ltime );
    outGuiFile << ltime << "\n";

     string st;
     string atType;
      // Parallel session detected
    if(type == "P")
     {
         atType = "Parallel Session";
         st = "There was a PARALLEL SESSION attack on the ";
     }
     // Replay attack detected
     else if(type == "R")
     {
         atType = "Replay";
         st = "There was a REPLAY attack on the ";
     }
     // single session attack detected
     else if(type == "S")
     {
         atType = "Single Session";
         st = "There was a SINGLE SESSION attack on the ";
     }
     // suspicious activity detected
     else if(type == "SS")
     {
         atType = "Suspicious behavior";
         st = "Suspicious behaviour was detected on the ";
     }
     char temp[30];
     st.append(protName);
     st.append(" protocol");

     st.append(" [Signature #");

     st.append(itoa(fsmNumber,temp,10));
     st.append("].");

     st.append("\n");
     st.append("This attack involved the following principals:\n");
     int i=1;
     map<string,string>::iterator pos;
     for(pos = ei.principals.begin(); pos != ei.principals.end(); ++pos)
     {
         st.append(itoa(i,temp,10));
         st.append(". ");
         st.append(pos->second);
         outGuiFile << pos->second << " ";
         st.append(": ");
         st.append(pos->first);
         outGuiFile << pos->first << " ";
         st.append("\n");
         i++;
     }
     outGuiFile << "!" << "\n";
     outGuiFile << atType << "\n";
     outGuiFile << fsmNumber << "\n" << "\n";

    outGuiFile.close();

    /*outReportFile << "*********************************************************\n";
    outReportFile << st  << '\n' << "Attack Detected on: " << ctime( &ltime ) << '\n';

     outReportFile.close();*/

     CloseHandle(fHandle);
     DeleteFile(lockFile);
     return 1;

 }

 /***************************************************************
  This function will enable the reporting of attack information
  such as abnormal termination or suspicious protocol activity
  by means of a text file. This file will have information like:
  Protocol name, type of attack, time of attack,parties involved.
  ***************************************************************/


 int writeSuspicious(ThreadParameters *tp,EventIndex ei,string protName,string type)
 {
```

```
char *lockFile = "lock.txt";
string atType;             // abnormal termination or suspicious behavior
HANDLE fHandle;

while(1)
{
    fHandle = (HANDLE) CreateFile(lockFile, GENERIC_READ, 0, NULL,
                             CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);


    if(fHandle == INVALID_HANDLE_VALUE)
    {
        EnterCriticalSection(&tp->gv.cout_cs);
         //cout << "File already locked" << endl;
         //Sleep(5000);
         LeaveCriticalSection(&tp->gv.cout_cs);
         CloseHandle(fHandle);

    }
    else
         break;
}

 //ofstream outReportFile(OUTFILE, ios::app);
 ofstream outGuiFile(OUTFILE_GUI,ios::app);
 /*if( !outReportFile )
 {
      cerr << "File could not be opened for writing!" << endl;

      CloseHandle(fHandle);
     DeleteFile(lockFile);
     return 0;
 }*/
 if(!outGuiFile)
 {
      cerr << "File could not be opened for writing!" << endl;

      CloseHandle(fHandle);
     DeleteFile(lockFile);
     return 0;
 }

 outGuiFile << "Attack" << "\n";
 outGuiFile << protName << "\n";


 time_t ltime;
 time( &ltime );
 outGuiFile << ltime << "\n";
 string st;

 // suspicious activity detected
 if(type == "SU")
 {
      st = "Suspicious behaviour was detected on the ";
      atType = "Suspicious behavior";
 }
 else if(type == "AB")
 {
      st = "Abnormal termination was detected on the ";
      atType = "Abnormal termination";
 }
 st.append(protName);
 st.append(" protocol");


 st.append("\n");
 st.append("This attack involved the following principals:\n");
 int i=1;
```

```
    map<string,string>::iterator pos;
    char temp[10];
    for(pos = ei.principals.begin(); pos != ei.principals.end(); ++pos)
    {
        st.append(itoa(i,temp,10));
        st.append(". ");
        st.append(pos->second);
        outGuiFile << pos->second << " ";
        st.append(": ");
        st.append(pos->first);
        outGuiFile << pos->first << " ";
        st.append("\n");
        i++;
    }
    outGuiFile << "!" << "\n";
    outGuiFile << atType << "\n";
    outGuiFile << "-1" << "\n" << "\n";

    /*outReportFile << "**********************************************************\n";
    outReportFile << st  << '\n' << "Attack Detected on: " << ctime( &ltime ) << '\n';
    outReportFile.close();*/
    outGuiFile.close();
    CloseHandle(fHandle);
    DeleteFile(lockFile);
    return 1;

}



/*******************************************
 Entry function for the threads monitoring
 attacks on distinct sessions.
 *******************************************/

unsigned __stdcall SessionAttackFunc (LPVOID lp)
{
  // Local variables
  int attack=0;
  vector<string> buffSig;
  DWORD thStatus;
  int vecLen=0;
  FSM *fs;

  int sigs=0, evtIndex=0;
  bool newEvent = true;
  bool evtSignal = false;
  buffSig.clear();
  ThreadInfo thInfo;
  DWORD currThreadId;

  currThreadId = GetCurrentThreadId();
  EventIndex ei;
  int suspicious = 0;
  int waitTime = 0;


  ThreadParameters *tp = (ThreadParameters*)lp;

  getThreadInfo(tp, &thInfo, currThreadId);

  ei.principals = thInfo.principals;
  ei.protocolName = thInfo.protName;
  ei.sessionNum = thInfo.sessionNumber;
  ei.sessionId = thInfo.sessionId;
  bool killThread = false;
  int numWaits = 0;
  while( 1 )
  {
      bool didTimeOut = false;
      int machineOver = 0;
```

```
thStatus = WaitForSingleObject(thInfo.threadEventHandle, 3000);

if(thStatus == WAIT_OBJECT_0)
{
  if(numWaits > 0)
       numWaits = 0;
  didTimeOut = false;
  evtSignal = true;
}

else if(thStatus == WSA_WAIT_TIMEOUT)
{
  didTimeOut = true;
  //cout << "IN TIMEOUT" << endl ;
 evtSignal = true;
  if(numWaits > 6)    // Timeout period == 18 seconds
  {
    killThread = true;
    break;
  }
}

else
{
   cout << " Problems!! " << endl;
    evtSignal = false;
}

if(evtSignal)
{
    while(1)
    {

         if(newEvent)
         {
             sigs = numSignatures(thInfo.protName.c_str(),&buffSig, &vecLen);
             EnterCriticalSection(&tp->gv.cout_cs);
             cout << "\nThere are "<< sigs << " signatures of the "
                   <<  thInfo.protName << " protocol." << endl;
             LeaveCriticalSection(&tp->gv.cout_cs);

             fs=new FSM[sigs];
             for(int j=0;j<sigs;j++)
                  fs[j].createMachine((j-1),buffSig, vecLen);
         }
         newEvent = false;
         //query monitor
         ei.vectorIndex = evtIndex;
         Event e = tp->mPtr->queryMonitor(&ei);

         if(e.type == "NoEvent")
         {
             evtSignal = false;
             if(didTimeOut)
             {
                numWaits++;
             }
             break;
         }
       machineOver = 0;
         for(int j=0;j<sigs;j++)
         {
            fs[j].advanceFSM(tp,e,ei);
            if(fs[j].stopFSM == 1)
            {
             machineOver++;
            }
         }
         if(machineOver == sigs)
```

```
              {
                  int regAttack = 0;
                  for(j=(sigs-1);j>=0;j--)
                  {
                     if(fs[j].detectType == "attackPP")
                     {
                        regAttack = 0;
                      attack = 3;
                        evtIndex++;
                        ei.vectorIndex = evtIndex;
                        break;
                     }
                     if(fs[j].detectType == "attackP")
                     {
                        regAttack = 0;
                      attack = 1;
                        evtIndex++;
                        ei.vectorIndex = evtIndex;
                        break;
                     }
                     else if(fs[j].detectType == "normal")
                     {
                         regAttack = 0;
                         attack = 2;
                         evtIndex++;
                         ei.vectorIndex = evtIndex;
                         break;
                     }
                     else if(fs[j].detectType == "attack")
                     {
                         regAttack = 1;
                         attack = 1;
                     }

                  }
                  if(attack == 1 && regAttack)
                  {
                       evtIndex ++;
                       ei.vectorIndex = evtIndex;

                  }
                  if(attack)
                          break;
                  else
                  {
                      suspicious = 1;
                  }

              }
              evtIndex++;



     }// end inner while

  } // end outer if
  if(attack)
       break;
}   // end outer while

if(suspicious && killThread)
{
    EnterCriticalSection(&tp->gv.cout_cs);
    cout << "****************************************************" << endl;
   cout << "There is a possibility of suspicious activity on the " << endl
         <<  thInfo.protName << " protocol." << endl;
   cout << "****************************************************" << endl << endl;
    LeaveCriticalSection(&tp->gv.cout_cs);
    tp->mPtr->deleteSession(&ei,"attackSessionOver");
   writeSuspicious(tp,ei,thInfo.protName,"SU");
}
```

```
  else if(killThread)
  {
      EnterCriticalSection(&tp->gv.cout_cs);
      cout << "*****************************************" << endl;
      cout << "Wait TIMEOUT expired!!" << endl
           << "Abnormal Termination was detected on the " << endl
           << thInfo.protName << " protocol." << endl;
      cout << "*****************************************" << endl << endl;
      LeaveCriticalSection(&tp->gv.cout_cs);


      tp->mPtr->deleteSession(&ei,"attackSessionOver");
      writeSuspicious(tp,ei,thInfo.protName,"AB");
  }
  else
  {
      if(attack == 3)
      {
          tp->mPtr->deleteSession(&ei,"attackParallelSessionOver");
      }
      if(attack == 1)
      {
          tp->mPtr->deleteSession(&ei,"attackSessionOver");
      }
      else if(attack == 2)
      {
          tp->mPtr->deleteSession(&ei,"normalSessionOver");
      }
  }
  EnterCriticalSection(&tp->gv.cout_cs);
  cout << "Thread just died!!" << endl;
  LeaveCriticalSection(&tp->gv.cout_cs);
  //cout << "Alive threads: " << tp->gv.tList.threadManager.size() << endl;
  removeThreadEntry(currThreadId,tp);
  return 0;

}


/**********************************************
  This function will retrieve the event handle
  for the particular thread, monitoring a session
  and protocol combination.

  Returns a HANDLE.
  *********************************************/


HANDLE getEventHandle (ThreadParameters *tp, EventIndex *ei)
{
     list<ThreadInfo>::iterator pos;
     HANDLE tHandle;
     EnterCriticalSection(&tp->gv.tList_cs);
     for(pos = tp->gv.tList.threadManager.begin(); pos != tp-
>gv.tList.threadManager.end(); ++pos)
     {
       if( !(pos -> protName.compare(ei->protocolName)) && (pos -> sessionNumber == ei-
>sessionNum) && (pos ->principals == ei->principals))
        {
             tHandle = pos -> threadEventHandle;
             break;
        }
     }
     LeaveCriticalSection(&tp->gv.tList_cs);

  return tHandle;

}

/************************************************************
 This function will return information about a thread and
 populate a ThreadInfo object, given a ThreadId.
 *************************************************************/


void getThreadInfo(ThreadParameters *tp, ThreadInfo *thInfo, DWORD thID)
{
     list<ThreadInfo>::iterator pos;
     EnterCriticalSection(&tp->gv.tList_cs);
     for(pos = tp->gv.tList.threadManager.begin(); pos != tp-
>gv.tList.threadManager.end(); ++pos)
    {
      if(pos -> threadEventHandle == tp->evtHandle )
       {
          pos -> threadId = thID;
          thInfo->principals = pos->principals;
          thInfo->protName = pos->protName;
          thInfo->sessionNumber = pos->sessionNumber;
          thInfo->sessionId = pos->sessionId;
          thInfo->threadEventHandle = pos->threadEventHandle;
          thInfo->threadId = pos->threadId;
          break;
       }
     }
    LeaveCriticalSection(&tp->gv.tList_cs);


}

/*************************************************************
 This function will remove the entry for a terminated thread
 from the tList vector which keeps track of all the alive
 threads.
 *************************************************************/


void removeThreadEntry(DWORD thId,ThreadParameters *tp)
{

    list<ThreadInfo>::iterator pos, pos1;
    EnterCriticalSection(&tp->gv.tList_cs);
     for(pos = tp->gv.tList.threadManager.begin(); pos != tp-
>gv.tList.threadManager.end(); ++pos)
    {

         if( pos ->threadId == thId )
         {
              tp->gv.tList.threadManager.erase(pos);
              break;
         }
     }
     LeaveCriticalSection(&tp->gv.tList_cs);

}






/****************************************************************************
 This function will read the KBASE of signatures, given a protocol Name
 and will populate a vector container (of strings) with all the attack/normal
 signatures corresponding to this protocol.
 ***************************************************************************/



int numSignatures(const char *prName, vector<string> *buffer, int *vecLen)
{
  char inBuffer[256], *token, *delimiter = " ";
  int endOfSig = 0;
  int numSig = 0;
  int i=0;

  ifstream fin;
  fin.open(KBASE);

  if(fin.fail())
  {
     cout << "Cannot open " << KBASE << endl;
     return 0;
  }


  while(!fin.eof() && !(endOfSig))
  {
     fin.getline(inBuffer,256,'\n');
    char *temp=new char[100];
     strcpy(temp,inBuffer);
     token = strtok(inBuffer, delimiter);

     if(strcmp(token,"begin") == 0)
     {
         token = strtok(NULL, delimiter);

         //read signatures for this protocol
         if(strcmp(token,prName) == 0)
         {
              buffer->push_back(temp);
              i++;
              while(1)
              {
                   fin.getline(inBuffer,256,'\n');
                   buffer->push_back(inBuffer);
                   i++;
                   token = strtok(inBuffer, delimiter);

                   if(strcmp(token,"end") == 0)
                   {

                       numSig++;
                       if(fin.eof())
                            break;

                       continue;
                   }

                   if(strcmp(token,"begin") == 0)
                   {
                               strtok(NULL, delimiter);
                       token = strtok(NULL, delimiter);

                       if(strcmp(token,"-1") == 0)
                       {
                            endOfSig = 1;
                            i--;
                            break;
                       }
                     continue;

                   }


              }

         }

     }


  }
```

```
  fin.close();
  *vecLen=i;
  return numSig;
}
```

```
/////////////////////////////
// SEADS_General.h
/////////////////////////////

#ifndef SEADS_GENERAL_H
#define SEADS_GENERAL_H

#define FD_SETSIZE 10000

#include <winsock2.h>
#include <iostream>
#include <fstream>

#include <string>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <set>

#include <process.h>

#include <cstdio>
#include <cstdarg>
#include <cstdlib>
#include <cstring>
#include <ctime>

using namespace std;

typedef map<string,string> nameMap;

class EventIndex
{
public:
    string protocolName;
    nameMap principals;                 //parties involved
    int sessionNum;                     //Id generated by monitor (sequential order)
    int sessionId;                      //Id generated by principals
    int vectorIndex;                //order in which session msgs are recv. Not
neccesarily the same as Event::msgSeqNum
    int vectorSize;                     //current size of vector...this is subject to
change
};


// Event class which corresponds to all the
// information for events reported to the IDE by the
// monitor. ( A send B 1  timestamp)

class Event
{
public:
    string p1;
    string type;
    string p2;
    int    msgSeqNum;
    time_t timeStamp;
};

//this class contains the number of sessions per group of principals
class PrincipalsSessions
{
public:
    nameMap principals;
    int     numSessions;
};


class Monitor
{
```

```
public:

    Monitor();
    ~Monitor();

    void  start();
    void  processMessages();
    void  welcomeMessage();
    void  printContents();
    void  storeEvent(Event*,EventIndex*);
    int   parseMessage(char*,Event*,EventIndex*);

    //Monitor's interface used by the IDE
    void  deleteSession(EventIndex*,string);
    Event queryMonitor(EventIndex *);
    vector<PrincipalsSessions> querySessions(string);

    DWORD  eventTotal;

    HANDLE hProducer,hConsumer,hIDE;
    unsigned int producerID, consumerID, ideID;
private:
    CRITICAL_SECTION cs;
    CRITICAL_SECTION monitorDB_CS;
};

#endif
```

```
//////////////////////////////////////////////////////////////
// Relevant parts of code from SEADS_Monitor.cpp
// (IDE – Monitor) interface code portions.
// Collaboration with Alex Melendez (Activity monitor project)
//////////////////////////////////////////////////////////////


void main()
{
    Monitor m;


    m.hProducer = (HANDLE) _beginthreadex(NULL,0,theProducer,(void *)&m,0,&m.producerID);
    m.hConsumer = (HANDLE) _beginthreadex(NULL,0,theConsumer,(void *)&m,0,&m.consumerID);
    m.hIDE      = (HANDLE) _beginthreadex(NULL,0,theIDE     ,(void *)&m,0,&m.ideID     );

    HANDLE threadList[3];
    threadList[0] = m.hProducer;
    threadList[1] = m.hConsumer;
    threadList[2] = m.hIDE;

    DWORD waitResult;

    waitResult = WaitForMultipleObjects(3,threadList,TRUE,INFINITE);

    if(waitResult == WAIT_OBJECT_0)
    {

        DWORD exitCode;
        GetExitCodeThread(m.hProducer,&exitCode);
        cout << "The producer has finished!" << endl;
        Sleep(10000);
    }

}


unsigned __stdcall theIDE(LPVOID lp)
{
    Monitor *m = (Monitor*)lp;

    // ** main.cpp declarations
    ThreadParameters tp;
    HANDLE hChildThread[1000];         // create handles for threads.
    //hChildThread = new HANDLE[10];
    DWORD waitResult;
    int i=0,dumpIndex=0;
    bool evtIndexSignal = false;
    Event e;
    EventIndex ei;
    // ** end of main.cpp declarations
    //DWORD threadId;

    Sleep(2000);
    m->welcomeMessage();

    cout<< "*************************************************\n"
           "**** Intrusion Detection Engine Operational  ****\n"
           "*************************************************\n"
        << endl;

    while(1)
    {
        waitResult = WaitForSingleObject(signalIDE, 10000);

        if(waitResult == WAIT_OBJECT_0)
        {
            evtIndexSignal = true;
        }

        else if(waitResult == WSA_WAIT_TIMEOUT)
        {

            EnterCriticalSection(&evtIndexCS);
            if(!evtIndexQueue.empty())
                evtIndexSignal = true;
            LeaveCriticalSection(&evtIndexCS);


            if(dumpIndex++ > 3)
            {
                m->printContents();
                EnterCriticalSection(&tp.gv.cout_cs);
                EnterCriticalSection(&tp.gv.tList_cs);
                cout << "Alive Threads : " << tp.gv.tList.threadManager.size();
                LeaveCriticalSection(&tp.gv.tList_cs);
                LeaveCriticalSection(&tp.gv.cout_cs);
                dumpIndex=0;
            }
        }

        if(evtIndexSignal)
        {

            while(1)
            {

                //need to read the queue
                EnterCriticalSection(&evtIndexCS);
                if(!evtIndexQueue.empty())
                {
                    ei = evtIndexQueue.front();
                    evtIndexQueue.pop();
                    e = m->queryMonitor(&ei);
                }
                else
                {
                    evtIndexSignal = false;
                    LeaveCriticalSection(&evtIndexCS);
                    break;
                }
                LeaveCriticalSection(&evtIndexCS);

                //if initial event
                if(e.msgSeqNum == 0)
                {
                     EnterCriticalSection(&tp.gv.cout_cs);
                     cout << "**** NEW SESSION ****" << endl;
                     LeaveCriticalSection(&tp.gv.cout_cs);

                     // Create the event kernel object which will be used
                     // to signal that thread in the future.
                     DWORD threadId;

                     tp.evtHandle = CreateEvent( NULL, FALSE, FALSE, NULL);
                     tp.mPtr = m;

                     ThreadInfo tInfo;
                     tInfo.threadEventHandle = tp.evtHandle;
                     tInfo.protName = ei.protocolName;
                     tInfo.sessionNumber = ei.sessionNum;
                     tInfo.sessionId = ei.sessionId;
                     tInfo.principals = ei.principals;

                     EnterCriticalSection(&tp.gv.tList_cs);
                     tp.gv.tList.threadManager.push_back(tInfo);
                     LeaveCriticalSection(&tp.gv.tList_cs);

                     // Create the session FSM thread
                     hChildThread[i++] = (HANDLE) _beginthreadex( NULL, 0,
SessionAttackFunc, (void *) &tp, 0, (unsigned *) &threadId);
                     /*if(i == 9)
                     {
                         delete [] hChildThread;
                         hChildThread = new HANDLE[10];
                         i = 0;

                     }*/
                     // Get the event handle to signal the relevent thread.
                     HANDLE tHandle = getEventHandle (&tp,&ei);
                     BOOL evtStatus = SetEvent(tHandle);

                     if(!evtStatus)
                        cout << "Unable to set manual event!!" << endl;

                     Sleep(50);
                }
                else
                {
                     //not a new session
                     HANDLE tHandle = getEventHandle (&tp, &ei);
                     BOOL evtStatus = SetEvent(tHandle);
                     Sleep(50);
                }
                /*** end of Sachin's code ***/
            }
        }


    }
    cout << "\n**the IDE has died**\n" << endl;
    return 0;
}
```

```
#if !defined(AFX_DIALOGS_H__5BD05A04_2B7F_4F1F_B059_D5161DF17CDC__INCLUDED_)
#define AFX_DIALOGS_H__5BD05A04_2B7F_4F1F_B059_D5161DF17CDC__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
///////////////////////////
// Dialogs.h : header file
//


struct comboHolder
{
 // Varibles to store previous combo box selection value.
    int n_FromApm;
    int n_FromDay;
    int n_FromHour;
    int n_FromMin;
    int n_FromMonth;
    int n_ToApm;
    int n_ToDay;
    int n_ToHour;
    int n_ToMin;
    int n_ToMonth;

};

// Arrays to fill up the combo boxes.

static CString months[12] =
{

    "Jan",
    "Feb",
    "Mar",
    "Apr",
    "May",
    "Jun",
    "Jul",
    "Aug",
    "Sep",
    "Oct",
    "Nov",
    "Dec"
};

/*static int hours[12] =
{
  1,2,3,4,5,6,7,8,9,10,11,12
};*/

static CString hours[12] =
{
  "01","02","03","04","05","06","07","08","09","10","11","12"
};

static int days[31] =
{
  1,2,3,4,5,6,7,8,9,10,
  11,12,13,14,15,16,17,18,19,20,
  21,22,23,24,25,26,27,28,29,30,
  31
};

/*static int mins[60] =
{
  01,02,03,04,05,06,07,08,09,10,
  11,12,13,14,15,16,17,18,19,20,
  21,22,23,24,25,26,27,28,29,30,
  31,32,33,34,35,36,37,38,39,40,
  41,42,43,44,45,46,47,48,49,50,
  51,52,53,54,55,56,57,58,59
};
*/

static CString mins[60] =
{
  "00","01","02","03","04","05","06","07","08","09","10",
  "11","12","13","14","15","16","17","18","19","20",
  "21","22","23","24","25","26","27","28","29","30",
  "31","32","33","34","35","36","37","38","39","40",
  "41","42","43","44","45","46","47","48","49","50",
  "51","52","53","54","55","56","57","58","59"
};

static CString apm[2] =
{
    "AM","PM"
};



/////////////////////////////////////////////////////////////////////////////
// CReportInfoDialog dialog

class CReportInfoDialog : public CDialog
{
// Construction
public:
    CReportInfoDialog(int fCall,struct comboHolder cHold, CWnd* pParent = NULL);   //
standard constructor

// Dialog Data
    //{{AFX_DATA(CReportInfoDialog)
    enum { IDD = IDD_REPORT_INFO };
    CComboBox    m_comboFromApm;
    CComboBox    m_comboToMonth;
    CComboBox    m_comboToMin;
    CComboBox    m_comboToHour;
    CComboBox    m_comboToDay;
    CComboBox    m_comboToApm;
    CComboBox    m_comboFromMonth;
    CComboBox    m_comboFromMin;
    CComboBox    m_comboFromHour;
    CComboBox    m_comboFromDay;
    CString  m_protName;
    //}}AFX_DATA

    // combo data holder variables
    CString m_FromApm;
    int m_FromMonth;
    int m_FromMin;
    int m_FromHour;
    int m_FromDay;
    CString m_ToApm;
    int m_ToMonth;
    int m_ToMin;
    int m_ToHour;
    int m_ToDay;

    int firstCall;
    struct comboHolder cHolder;




// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CReportInfoDialog)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CReportInfoDialog)
    virtual BOOL OnInitDialog();
    afx_msg void OnSelchangeCombo();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_DIALOGS_H__5BD05A04_2B7F_4F1F_B059_D5161DF17CDC__INCLUDED_)
```

```
// Dialogs.cpp : implementation file
//

#include "stdafx.h"
#include "IDE.h"
#include "Dialogs.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//#include "IDEDoc.h"

/////////////////////////////////////////////////////////////////////////////
// CReportInfoDialog dialog


CReportInfoDialog::CReportInfoDialog(int fCall,struct comboHolder cHold, CWnd* pParent
/*=NULL*/)
     : CDialog(CReportInfoDialog::IDD, pParent)
{
     //{{AFX_DATA_INIT(CReportInfoDialog)
     m_protName = _T("");
     //}}AFX_DATA_INIT
     firstCall = fCall;
    cHolder = cHold;

}


void CReportInfoDialog::DoDataExchange(CDataExchange* pDX)
{
     CDialog::DoDataExchange(pDX);
     //{{AFX_DATA_MAP(CReportInfoDialog)
     DDX_Control(pDX, IDC_FROM_APM_COMBO, m_comboFromApm);
     DDX_Control(pDX, IDC_UPTO_MONTH_COMBO, m_comboToMonth);
     DDX_Control(pDX, IDC_UPTO_MIN_COMBO, m_comboToMin);
     DDX_Control(pDX, IDC_UPTO_HOUR_COMBO, m_comboToHour);
     DDX_Control(pDX, IDC_UPTO_DAY_COMBO, m_comboToDay);
     DDX_Control(pDX, IDC_UPTO_APM_COMBO, m_comboToApm);
     DDX_Control(pDX, IDC_FROM_MONTH_COMBO, m_comboFromMonth);
     DDX_Control(pDX, IDC_FROM_MIN_COMBO, m_comboFromMin);
     DDX_Control(pDX, IDC_FROM_HOUR_COMBO, m_comboFromHour);
     DDX_Control(pDX, IDC_FROM_DAY_COMBO, m_comboFromDay);
     DDX_Text(pDX, IDC_PROTOCOL_EDIT, m_protName);
     DDV_MaxChars(pDX, m_protName, 70);
     //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CReportInfoDialog, CDialog)
     //{{AFX_MSG_MAP(CReportInfoDialog)
     ON_CBN_SELCHANGE(IDC_FROM_APM_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_FROM_DAY_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_FROM_HOUR_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_FROM_MIN_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_FROM_MONTH_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_UPTO_APM_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_UPTO_DAY_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_UPTO_HOUR_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_UPTO_MIN_COMBO, OnSelchangeCombo)
     ON_CBN_SELCHANGE(IDC_UPTO_MONTH_COMBO, OnSelchangeCombo)
     //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CReportInfoDialog message handlers
```

```
BOOL CReportInfoDialog::OnInitDialog()
{

    CDialog::OnInitDialog();
    char str[30];

    // TODO: Add extra initialization here
    // Load the months of the year into the combo box.
    for(int i=0; i<12; i++)
    {
        m_comboFromMonth.AddString(months[i]);
        m_comboToMonth.AddString(months[i]);
    }
    // Load the days of the month into the combo box.
    for(i=0; i<31; i++)
    {
        m_comboFromDay.AddString(itoa(days[i],str,10));
        m_comboToDay.AddString(itoa(days[i],str,10));
    }
    // Load the hours into the combo box.
    for(i=0; i<12; i++)
    {
        m_comboFromHour.AddString(hours[i]);
        m_comboToHour.AddString(hours[i]);
    }
    // Load the minutes into the combo box.
    for(i=0; i<60; i++)
    {
        m_comboFromMin.AddString(mins[i]);
        m_comboToMin.AddString(mins[i]);
    }
    // Load AM/PM into the combo box.
    for(i=0; i<2; i++)
    {
        m_comboFromApm.AddString(apm[i]);
        m_comboToApm.AddString(apm[i]);
    }

    // Initial Display
    if(firstCall == 0)
    {
             //firstCall = 1;
             m_comboFromMonth.SetCurSel(0);
             m_comboToMonth.SetCurSel(0);
             m_comboFromDay.SetCurSel(0);
             m_comboToDay.SetCurSel(0);
             m_comboFromMin.SetCurSel(0);
             m_comboToMin.SetCurSel(0);
             m_comboFromApm.SetCurSel(0);
             m_comboToApm.SetCurSel(0);
             m_comboFromHour.SetCurSel(0);
             m_comboToHour.SetCurSel(0);

             // Initializations of dialog box variables (attached to combo box controls)
             m_FromApm = apm[m_comboFromApm.GetCurSel()];
             m_FromMonth = m_comboFromMonth.GetCurSel();
             m_FromDay = days[m_comboFromDay.GetCurSel()];
             m_FromHour = atoi((const char *) hours[m_comboFromHour.GetCurSel()]);
             m_FromMin = atoi((const char *) mins[m_comboFromMin.GetCurSel()]);
             m_ToApm = apm[m_comboToApm.GetCurSel()];
             m_ToMonth = m_comboToMonth.GetCurSel();
             m_ToDay = days[m_comboToDay.GetCurSel()];
             m_ToHour = atoi((const char *) hours[m_comboToHour.GetCurSel()]);
             m_ToMin = atoi((const char *) mins[m_comboToMin.GetCurSel()]);
    }
    else
    {
             m_comboFromMonth.SetCurSel(cHolder.n_FromMonth);
             m_comboToMonth.SetCurSel(cHolder.n_ToMonth);
             m_comboFromDay.SetCurSel(cHolder.n_FromDay);
             m_comboToDay.SetCurSel(cHolder.n_ToDay);
```

```
            m_comboFromMin.SetCurSel(cHolder.n_FromMin);
            m_comboToMin.SetCurSel(cHolder.n_ToMin);
            m_comboFromApm.SetCurSel(cHolder.n_FromApm);
            m_comboToApm.SetCurSel(cHolder.n_ToApm);
            m_comboFromHour.SetCurSel(cHolder.n_FromHour);
            m_comboToHour.SetCurSel(cHolder.n_ToHour);

            // Initializations of dialog box variables (attached to combo box controls)
            m_FromApm = apm[m_comboFromApm.GetCurSel()];
            m_FromMonth = m_comboFromMonth.GetCurSel();
            m_FromDay = days[m_comboFromDay.GetCurSel()];
            m_FromHour = atoi((const char *) hours[m_comboFromHour.GetCurSel()]);
            m_FromMin = atoi((const char *) mins[m_comboFromMin.GetCurSel()]);
            m_ToApm = apm[m_comboToApm.GetCurSel()];
            m_ToMonth = m_comboToMonth.GetCurSel();
            m_ToDay = days[m_comboToDay.GetCurSel()];
            m_ToHour = atoi((const char *) hours[m_comboToHour.GetCurSel()]);
            m_ToMin = atoi((const char *) mins[m_comboToMin.GetCurSel()]);
    }

    return TRUE;  // return TRUE unless you set the focus to a control
                  // EXCEPTION: OCX Property Pages should return FALSE
}

void CReportInfoDialog::OnSelchangeCombo()
{
    // TODO: Add your control notification handler code here
        //m_protName = dlg.m_protName;



        m_FromApm = apm[m_comboFromApm.GetCurSel()];
        m_FromMonth = m_comboFromMonth.GetCurSel();
        m_FromDay = days[m_comboFromDay.GetCurSel()];
        m_FromHour = atoi((const char *) hours[m_comboFromHour.GetCurSel()]);
        m_FromMin = atoi((const char *) mins[m_comboFromMin.GetCurSel()]);
        m_ToApm = apm[m_comboToApm.GetCurSel()];
        m_ToMonth = m_comboToMonth.GetCurSel();
        m_ToDay = days[m_comboToDay.GetCurSel()];
        m_ToHour = atoi((const char *) hours[m_comboToHour.GetCurSel()]);
        m_ToMin = atoi((const char *) mins[m_comboToMin.GetCurSel()]);



        cHolder.n_FromMonth = m_comboFromMonth.GetCurSel();
        cHolder.n_ToMonth =     m_comboToMonth.GetCurSel();
        cHolder.n_FromDay = m_comboFromDay.GetCurSel();
        cHolder.n_ToDay = m_comboToDay.GetCurSel();
        cHolder.n_FromMin = m_comboFromMin.GetCurSel();
        cHolder.n_ToMin = m_comboToMin.GetCurSel();
        cHolder.n_FromApm = m_comboFromApm.GetCurSel();
        cHolder.n_ToApm = m_comboToApm.GetCurSel();
        cHolder.n_FromHour = m_comboFromHour.GetCurSel();
        cHolder.n_ToHour = m_comboToHour.GetCurSel();
}
```

```
// IDEDoc.h : interface of the CIDEDoc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_IDEDOC_H__B57ECC08_4DF7_4EEF_8125_60D8E4FDEFE7__INCLUDED_)
#define AFX_IDEDOC_H__B57ECC08_4DF7_4EEF_8125_60D8E4FDEFE7__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Dialogs.h"
#include "BackupDialog.h"
#include <fstream>

#include <vector>
#include <string>
#include <map>
#include <time.h>

#define INFILE "report.txt"
#define BACKUPFILE "backup.txt"


using namespace std;

// This class holds the members which will be used to display lines of
// output to the GUI

class Disp : public CObject
{
public:
      // Methods
      Disp();
      virtual void Serialize(CArchive &ar);

      // Data members
      CString attack;
      CString involves;
      CString principals;
      CString blankLine;
      CString time;
};


// This class contains all the information about a single attack
// The active attack report file is read into objects of this class
// one by one.

class AttackInfo
{
public:
    AttackInfo();

    CString protName;    // protocol on which the attack takes place.
    int attacktime;      // time of the attack
    CString attackType;  // type of attack - single session/ replay etc.
    int fsmNumber;       // attack signature number
    map<string,string> nameMap;  // principals involved
};


class CIDEDoc : public CDocument
{
protected: // create from serialization only
    CIDEDoc();
    DECLARE_DYNCREATE(CIDEDoc)

// Attributes
public:
    // variables to hold the dialog box input from the user
    CString m_protName;   // protocol name field input
    CString m_display;    // display string
    CString m_welcome;    // string for the welcome message
    CString m_FromApm;    // AM/PM selection
    int m_FromDay;        // Day of the month selection
    int m_FromHour;
    int m_FromMin;
    int m_FromMonth;
    CString m_ToApm;
    int m_ToDay;
    int m_ToHour;
    int m_ToMin;
    int m_ToMonth;
    int firstCreateDialog;
    struct comboHolder cHolder; // structure to hold previous selections of the combo
box.

    int displayNum;      // 0 if welcome message to be displayed, 1 otherwise
    vector<AttackInfo> reportData;   // vector to hold the attack info. from file.
    time_t initTime;
    time_t finalTime;
    int numAttacks;
    CString openLine;
    Disp *display; // data class to hold the report output to be displayed

// Operations
public:
    int ReadAttackFile(CString);
    void CreateDisplay(Disp *,int);
    void ConstructTime();
    void fileBackup();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CIDEDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
    virtual void DeleteContents();
    virtual void OnCloseDocument();
    //}}AFX_VIRTUAL

// Implementation
public:
    void GetDocSizes(int,CSize&,CSize&,CSize&);
    virtual ~CIDEDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CIDEDoc)
    afx_msg void OnToolReport();
    afx_msg void OnMenuBackup();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_IDEDOC_H__B57ECC08_4DF7_4EEF_8125_60D8E4FDEFE7__INCLUDED_)
```

```
// IDEDoc.cpp : implementation of the CIDEDoc class
//

#include "stdafx.h"
#include "IDE.h"

#include "IDEDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW

#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CIDEDoc

IMPLEMENT_DYNCREATE(CIDEDoc, CDocument)

BEGIN_MESSAGE_MAP(CIDEDoc, CDocument)
    //{{AFX_MSG_MAP(CIDEDoc)
    ON_COMMAND(ID_TOOL_REPORT, OnToolReport)
    ON_COMMAND(ID_MENU_CREATE, OnToolReport)
    ON_COMMAND(ID_MENU_BACKUP, OnMenuBackup)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CIDEDoc construction/destruction

CIDEDoc::CIDEDoc()
{
    // TODO: add one-time construction code here
    m_welcome = "Welcome to the IDE Report Generator!";
    m_display = " ";
    displayNum = 0;
    numAttacks = 0;
    openLine = " ";
    cHolder.n_FromApm = 0;
    cHolder.n_FromDay = 0;
    cHolder.n_FromHour = 0;
    cHolder.n_FromMin = 0;
    cHolder.n_FromMonth = 0;
    cHolder.n_ToApm = 0;
    cHolder.n_ToDay = 0;
    cHolder.n_ToHour = 0;
    cHolder.n_ToMin = 0;
    cHolder.n_ToMonth = 0;
    firstCreateDialog = 0;
}

CIDEDoc::~CIDEDoc()
{
}

BOOL CIDEDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}



/////////////////////////////////////////////////////////////////////////////
// CIDEDoc serialization
```

```
void CIDEDoc::Serialize(CArchive& ar)
{

}

/////////////////////////////////////////////////////////////////////////////
// CIDEDoc diagnostics

#ifdef _DEBUG
void CIDEDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CIDEDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CIDEDoc commands


// Handler when the user hits the create report button.
void CIDEDoc::OnToolReport()
{
    // TODO: Add your command handler code here
    DeleteContents();
    CReportInfoDialog dlg(firstCreateDialog,cHolder);
    firstCreateDialog = 1;
    dlg.m_protName = m_protName;


    ofstream fout;
    fout.open("debug.txt");
    int modal = dlg.DoModal();

    if(modal == IDCANCEL)
    {
        displayNum = 0;
         UpdateAllViews(NULL);
         return;
    }
    else if(modal == IDOK)
    {
       // Retrieve value entered by the user
         cHolder = dlg.cHolder;
         m_protName = dlg.m_protName;
         m_FromMonth = dlg.m_FromMonth;

         m_FromApm = dlg.m_FromApm;
         m_FromDay = dlg.m_FromDay;
         m_FromHour = dlg.m_FromHour;
         m_FromMin = dlg.m_FromMin;
         m_ToApm = dlg.m_ToApm;
         m_ToDay = dlg.m_ToDay;
         m_ToHour = dlg.m_ToHour;
         m_ToMin = dlg.m_ToMin;
         m_ToMonth = dlg.m_ToMonth;

         fout << "From Month: " << m_FromMonth << endl;
         fout << "From Day: " << m_FromDay << endl;
         fout << "To Month: " << m_ToMonth << endl;
         fout << "To Day: " << m_ToDay << endl;

    }
    fout.close();
    ConstructTime();
    numAttacks = ReadAttackFile(m_protName);
    display = new Disp[numAttacks];

    if(numAttacks >0)
    {
        openLine = "Attack Report";
         CreateDisplay(display,numAttacks);
    }
    else
         m_display = "There are no attacks at this time matching your specifications!!";

    displayNum = 1;
    UpdateAllViews(NULL);


}


/////////////////////////////////////////////////////////////////
/// readAttackFile
/// This function reads the attack Report file and
/// extracts relevant information to be displayed to the GUI
/////////////////////////////////////////////////////////////////


int CIDEDoc::ReadAttackFile(CString protName)
{
  // File locking code follows
   char *lockFile = "lock.txt";
   HANDLE fHandle;

   while(1)
   {
            fHandle = (HANDLE) CreateFile(lockFile, GENERIC_READ, 0, NULL,
                                     CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
            if(fHandle == INVALID_HANDLE_VALUE)
            {
                DWORD gle = GetLastError();
                CloseHandle(fHandle);
            }
            else
                break;
    }
    // End of file locking code
    HANDLE fileHandle;
    fileHandle = (HANDLE) CreateFile(INFILE,GENERIC_READ,0,NULL,OPEN_EXISTING,
                                      FILE_ATTRIBUTE_NORMAL,NULL);
    if(fileHandle == INVALID_HANDLE_VALUE)
    {
         return -1;
    }
    DWORD fileSizeLow, fileHigh,*fileSizeHigh;
    fileSizeHigh = &fileHigh;
    fileSizeLow = GetFileSize(fileHandle,fileSizeHigh);
    ofstream fout;
    fout.open("debug.txt",ios::app);
    fout << fileSizeLow << endl;
    fout << *fileSizeHigh << endl;
    fout.close();
    CloseHandle(fileHandle);
    if(fileSizeLow > 20000)
         fileBackup();


    ifstream inReportFile(INFILE, ios::in);

    if( !inReportFile )
    {
         //cerr << "File could not be opened for writing!" << endl;
        return -1;
    }
    // parse the file contents for relevant information
```

```
int i=0;
char *delimiter = " ";
string smallName, longName;
char buffer[100];
reportData.clear();
// Read from the active SEADS file viz "report.txt"

    while(!inReportFile.eof())
    {

        inReportFile.getline(buffer,60,'\n');
        if(strcmp(buffer,"Attack") == 0)
        {
            AttackInfo ai;
            inReportFile.getline(buffer,60,'\n');
            if(protName == buffer || protName == "ALL")
            {
                ai.protName =  buffer;
                inReportFile.getline(buffer,60,'\n');
                ai.attacktime = atoi(buffer);
                inReportFile.getline(buffer,60,'\n');
                smallName = strtok(buffer,delimiter);
                longName = strtok(NULL,delimiter);

ai.nameMap.insert(map<string,string>::value_type(smallName,longName));
                smallName = strtok(NULL, delimiter);
                while(smallName != "!")
                {
                    longName = strtok(NULL, delimiter);

ai.nameMap.insert(map<string,string>::value_type(smallName,longName));
                    smallName = strtok(NULL, delimiter);

                }
                inReportFile.getline(buffer,60,'\n');
                ai.attackType = buffer;
                inReportFile.getline(buffer,60,'\n');
                ai.fsmNumber = atoi(buffer);
                if(ai.attacktime <= finalTime && ai.attacktime >= initTime)
                {
                  reportData.push_back(ai);
                i++;
                }
            }

        }
    }
CloseHandle(fHandle);
DeleteFile(lockFile);


// Read from the back up file "backup.txt"
ifstream inBackupFile(BACKUPFILE,ios::in);

if( !inBackupFile )
{
    //cerr << "File could not be opened for writing!" << endl;
   return -1;
}

while(!inBackupFile.eof())
    {

        inBackupFile.getline(buffer,60,'\n');
        if(strcmp(buffer,"Attack") == 0)
        {
            AttackInfo ai;
            inBackupFile.getline(buffer,60,'\n');
            if(protName == buffer || protName == "ALL")
            {
                ai.protName =  buffer;
```

```
                       inBackupFile.getline(buffer,60,'\n');
                       ai.attacktime = atoi(buffer);
                       inBackupFile.getline(buffer,60,'\n');
                       smallName = strtok(buffer,delimiter);
                       longName = strtok(NULL,delimiter);

     ai.nameMap.insert(map<string,string>::value_type(smallName,longName));
                       smallName = strtok(NULL, delimiter);
                       while(smallName != "!")
                       {
                            longName = strtok(NULL, delimiter);

     ai.nameMap.insert(map<string,string>::value_type(smallName,longName));
                            smallName = strtok(NULL, delimiter);

                       }
                       inBackupFile.getline(buffer,60,'\n');
                       ai.attackType = buffer;
                       inBackupFile.getline(buffer,60,'\n');
                       ai.fsmNumber = atoi(buffer);
                       if(ai.attacktime <= finalTime && ai.attacktime >= initTime)
                       {
                         reportData.push_back(ai);
                      i++;
                       }
                  }

             }
         }


    return i;
}

////////////////////////////////////////////////////////////////
// This function will construct the time in seconds elapsed
// since January 1 1970, based
// upon the users input for day/month/hour/minutes and seconds.
////////////////////////////////////////////////////////////////

void CIDEDoc::ConstructTime()
{

 struct tm timestruct;
 if(m_FromApm == "PM")
 {
   if(m_FromHour == 12)
         timestruct.tm_hour = 0;
   else
        timestruct.tm_hour = m_FromHour + 12;
 }

 else
      timestruct.tm_hour = m_FromHour;
 timestruct.tm_isdst = -1;
 timestruct.tm_mday = m_FromDay;
 timestruct.tm_min = m_FromMin;
 timestruct.tm_mon = m_FromMonth;
 timestruct.tm_sec = 0;

 timestruct.tm_year = 101;

 initTime = mktime(&timestruct);
 struct tm timestruct1;
 if(m_ToApm == "PM")
 {
   timestruct1.tm_hour = m_ToHour + 12;
 }
 else
      timestruct1.tm_hour = m_ToHour;
 timestruct1.tm_isdst = -1;
 timestruct1.tm_mday = m_ToDay;
 timestruct1.tm_min = m_ToMin;
 timestruct1.tm_mon = m_ToMonth;
 timestruct1.tm_sec = 0;
 timestruct1.tm_year = 101;

 finalTime = mktime(&timestruct1);
}


///////////////////////////////////////////////////////////////////
//This function will create the GUI attack report display to show
//the attack information.
///////////////////////////////////////////////////////////////////
void CIDEDoc::CreateDisplay(Disp *d,int i)
{


  for(int j=0; j<i; j++)
  {
    char str[20];
    d[j].attack += itoa(j+1,str,10);
    d[j].attack += ". ";
    if(reportData[j].attackType == "Suspicious behavior" || reportData[j].attackType ==
         "Abnormal termination")
    {
         d[j].attack += reportData[j].attackType + " was detected on the ";

    }
    else
    {
         d[j].attack += "A " + reportData[j].attackType + " attack was detected on the ";
    }

    d[j].attack += reportData[j].protName + " protocol.";
    char temp[30];

    if(reportData[j].fsmNumber >=0)
    {
         CString num = itoa(reportData[j].fsmNumber,temp,10);
        d[j].attack += " (Signature #" + num + ")";

    }

    d[j].involves += "    The following principals were involved: ";

    int k=1;
    map<string,string>::iterator pos;
    for(pos = reportData[j].nameMap.begin(); pos != reportData[j].nameMap.end(); ++pos)
    {
         d[j].principals += "    ";
         d[j].principals += itoa(k,str,10);
         d[j].principals += ".   ";
         d[j].principals +=  pos->first.c_str();
         d[j].principals += ":   ";
         d[j].principals +=  pos->second.c_str();
         d[j].principals += ".";
         k++;
    }
    d[j].blankLine = " ";
    int timeSec = reportData[j].attacktime;
    struct tm *tm1 = localtime((const long *) &timeSec);

    d[j].time += "    Attack Time: ";
    d[j].time += asctime(tm1);



  }

}
```

```
///////////////////////////////////////////////////////
//Handler for the action the user takes by hitting the
//menu backup button. This button will back up the
//active attack report file.
///////////////////////////////////////////////////////
void CIDEDoc::OnMenuBackup()
{
     // TODO: Add your command handler code here

     // Handler for backing up the attack report file.
     CBackupDialog cdlg;
     if(cdlg.DoModal()==IDOK)
     {
          displayNum = 0;
          UpdateAllViews(NULL);
        // do nothing
     }
    // File backup code written below.
     char *lockFile = "lock.txt";
     HANDLE fHandle;

        while(1)
        {
            fHandle = (HANDLE) CreateFile(lockFile, GENERIC_READ, 0, NULL,
                                     CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
            if(fHandle == INVALID_HANDLE_VALUE)
            {
                //cout << "File already locked" << endl;
                CloseHandle(fHandle);
            }
            else
                break;
        }
         // End of file locking code

    fileBackup();
     CloseHandle(fHandle);
     DeleteFile(lockFile);

         // File locking code follows

}

//////////////////////////////////////////////////////////////
//Function which results in the backing up of the active attack
//report file "report.txt" to another file "backup.txt"
//////////////////////////////////////////////////////////////
void CIDEDoc::fileBackup()
{

    HANDLE hSrc, hDest;

    DWORD dwRead, dwWritten;

    char pBuffer[1024];



    hSrc = CreateFile(INFILE, GENERIC_READ, 0, NULL,

                      OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);


    hDest = CreateFile(BACKUPFILE, GENERIC_WRITE, 0, NULL,

                       CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

    do

    {
        ReadFile(hSrc, pBuffer, sizeof(pBuffer), &dwRead, NULL);

        if (dwRead != 0)

            WriteFile(hDest, pBuffer, dwRead, &dwWritten, NULL);

    } while (dwRead != 0);

    CloseHandle(hSrc);
    CloseHandle(hDest);
    ofstream fout;
    time_t ltime;
    time(&ltime);

    fout.open(INFILE,ios::out);
    fout << "Last backup on ";
    fout << ctime(&ltime) << "\n\n" ;
    fout.close();




}

void CIDEDoc::GetDocSizes(int m_HeightLine, CSize &sizeTotal, CSize &sizePage, CSize
&sizeLine)
{
  int nHeight;
  int nWidth;
  const int FUDGE_H = 400;
  const int FUDGE_V = 400;
  if(displayNum == 0)
  {
     nHeight = 200 + FUDGE_H;
      nWidth  =  200 + FUDGE_V;
      sizeTotal = CSize(nWidth, nHeight);
  }
  else
  {
      nHeight = numAttacks * 10 * m_HeightLine + FUDGE_H;
      nWidth = 200 + FUDGE_V;
      sizeTotal = CSize(nWidth, nHeight);
  }

  sizePage = CSize(sizeTotal.cx /10, sizeTotal.cy /10);
  sizeLine = CSize(sizePage.cx /10, sizePage.cy /10);

}



AttackInfo::AttackInfo()
{

    //do nothing
}


////////////////////////////////////////////////
// Disp class implementation

//IMPLEMENT_SERIAL(Disp, CObject, 1)

Disp::Disp()
{

  // do nothing

}
```

```
void Disp::Serialize(CArchive &ar)
{
 // do nothing
}

BOOL CIDEDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;

    // TODO: Add your specialized creation code here

    return TRUE;
}


void CIDEDoc::DeleteContents()
{
    // TODO: Add your specialized code here and/or call the base class

}

void CIDEDoc::OnCloseDocument()
{
    // TODO: Add your specialized code here and/or call the base class

    CDocument::OnCloseDocument();

}
```

```
// IDEView.h : interface of the CIDEView class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_IDEVIEW_H__53E7EC96_D841_45F9_A4CD_769E9D2BAB29__INCLUDED_)
#define AFX_IDEVIEW_H__53E7EC96_D841_45F9_A4CD_769E9D2BAB29__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CIDEView : public CScrollView
{
protected: // create from serialization only
     CIDEView();
     DECLARE_DYNCREATE(CIDEView)

// Attributes
public:
     CIDEDoc* GetDocument();
     int m_nHeightLine;
     int m_nPageWidth;
     int m_nPageHeight;
     int m_nMapMode;

// Operations
public:

// Overrides
     // ClassWizard generated virtual function overrides
     //{{AFX_VIRTUAL(CIDEView)
     public:
     virtual void OnDraw(CDC* pDC);  // overridden to draw this view
     virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
     virtual void OnInitialUpdate();
     virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
     protected:
     virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
     virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
     virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
     virtual void OnPrint(CDC* pDC, CPrintInfo* pInfo);
     //}}AFX_VIRTUAL

// Implementation
public:
     virtual ~CIDEView();
#ifdef _DEBUG
     virtual void AssertValid() const;
     virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
     //{{AFX_MSG(CIDEView)
          // NOTE - the ClassWizard will add and remove member functions here.
          //    DO NOT EDIT what you see in these blocks of generated code !
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in IDEView.cpp
inline CIDEDoc* CIDEView::GetDocument()
   { return (CIDEDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
```

```
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_IDEVIEW_H__53E7EC96_D841_45F9_A4CD_769E9D2BAB29__INCLUDED_)
```

```
// IDEView.cpp : implementation of the CIDEView class
//

#include "stdafx.h"
#include "IDE.h"

#include "IDEDoc.h"
#include "IDEView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CIDEView

IMPLEMENT_DYNCREATE(CIDEView, CScrollView)

BEGIN_MESSAGE_MAP(CIDEView, CScrollView)
    //{{AFX_MSG_MAP(CIDEView)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CIDEView construction/destruction

CIDEView::CIDEView()
{
    // TODO: add construction code here

}

CIDEView::~CIDEView()
{
}

BOOL CIDEView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs

    return CScrollView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CIDEView drawing

const int MARGIN_LEFT = 30;
const int LINES = 30;

void CIDEView::OnDraw(CDC* pDC)
{
    CIDEDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CSize sizeTotal,sizePage,sizeLine;
    sizeTotal = sizePage = sizeLine = CSize(0,0);

    // TODO: add draw code for native data here
    if(pDoc->displayNum == 0)
    {

        pDoc->GetDocSizes(m_nHeightLine, sizeTotal, sizePage, sizeLine);
        SetScrollSizes(MM_TEXT, sizeTotal, sizePage,sizeLine);
       //pDC->TextOut(100,100,pDoc->m_welcome);
        CBitmap bitmap;
        CDC dcMemory;
        bitmap.LoadBitmap(IDB_BITMAP1);
        dcMemory.CreateCompatibleDC(pDC);
        dcMemory.SelectObject(&bitmap);

        pDC->BitBlt(250,100,500,500,&dcMemory,0,0,SRCCOPY);
    }
    else
    {

          ASSERT(m_nHeightLine > 0);

          int nLines = 5;
          if(pDoc -> numAttacks <=0)
          {
              pDoc->GetDocSizes(m_nHeightLine, sizeTotal, sizePage, sizeLine);
              SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage,sizeLine);
              pDC->TextOut(MARGIN_LEFT, -nLines * m_nHeightLine, pDoc->m_display);
          }
          else
          {
              pDoc->GetDocSizes(m_nHeightLine, sizeTotal, sizePage, sizeLine);
              SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage,sizeLine);
              pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine, pDoc->openLine);
              for(int i=0;i < pDoc ->numAttacks;i++)
              {

                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].blankLine);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].attack);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].blankLine);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].involves);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].blankLine);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].principals);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].blankLine);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].blankLine);
                  pDC->TextOut(MARGIN_LEFT, -nLines++ * m_nHeightLine,
                  pDoc->display[i].time);
                  nLines++;
              }
          }

    }
}

//////////////////////////////////////////////////////////////////////////////
// CIDEView printing

BOOL CIDEView::OnPreparePrinting(CPrintInfo* pInfo)
{
    /////////////////////
    CIDEDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CSize sizeTotal, sizePage, sizeLine;
    sizeTotal = sizePage = sizeLine = CSize(0,0);

    pDoc->GetDocSizes(m_nHeightLine,sizeTotal,sizePage,sizeLine);

    int numPages = (int) sizeTotal.cy/(m_nPageHeight);

    if(numPages < 1)
```

```
        numPages = 1;

    pInfo->SetMaxPage(numPages);

    // default preparation
    return DoPreparePrinting(pInfo);
}

void CIDEView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: add extra initialization before printing
    // Get the printer's resolution in millimeters
    int nHorzSize = pDC->GetDeviceCaps(HORZSIZE);
    int nVertSize = pDC->GetDeviceCaps(VERTSIZE);

    m_nPageWidth = (double)nHorzSize / 25.4 * 100.0;
    TRACE("m_nPageWidth = %d\n",m_nPageWidth);

    m_nPageHeight = (double)nVertSize /25.4 * 100.0;
    TRACE("m_nPageHeight = %d\n",m_nPageHeight);

}

void CIDEView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
// CIDEView diagnostics

#ifdef _DEBUG
void CIDEView::AssertValid() const
{
    CScrollView::AssertValid();
}

void CIDEView::Dump(CDumpContext& dc) const
{
    CScrollView::Dump(dc);
}

CIDEDoc* CIDEView::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CIDEDoc)));
    return (CIDEDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CIDEView message handlers
void CIDEView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    // TODO: Add your specialized code here and/or call the base class
    CClientDC dc(this);
    // Declare a TEXTMETRIC variable.
    TEXTMETRIC tm;

    // fill up the variable info.
    dc.GetTextMetrics(&tm);
    m_nHeightLine = tm.tmHeight + tm.tmExternalLeading;
    m_nMapMode = MM_LOENGLISH;
    CSize sizeTotal,sizePage,sizeLine;
    sizeTotal = sizePage = sizeLine = CSize(0,0);



    CIDEDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    pDoc->GetDocSizes(m_nHeightLine,sizeTotal,sizePage,sizeLine);

    SetScrollSizes(m_nMapMode, sizeTotal, sizePage,sizeLine);

}


void CIDEView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class

    CScrollView::OnPrepareDC(pDC, pInfo);
    if(pDC->IsPrinting())
    {
        int nPages = pInfo->m_nCurPage - 1;
        int y = (nPages) * -m_nPageHeight;

        pDC->SetWindowOrg(0, y);
    }
}

void CIDEView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    ASSERT_VALID(pDC);
    {

     }
    CScrollView::OnPrint(pDC, pInfo);
}
```

11. References

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

[2] H.Debar, M.Dacier, A.Wespi, “Towards a Taxonomy of Intrusion Detection Systems”, Elsevier Science B.V 31 (1999) 805-822

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

[4] Dorothy Denning and G.Sacco, “Timestamps in Key Distribution Protocols”, Communications of the ACM, 24(8), August 1981, pp. 533-534.

[5] Roger M. Needham and Michael Schroeder, “Using Encryption for Authentication in Large Networks of Computers”, Communications of the ACM, 21(12), December 1978, pp. 994-995.

[6] Alec Yasinsac, “Detecting Intrusions in Security Protocols”, Proceedings of First Workshop on Intrusion Detection Systems, in the 7th ACM Conference on Computer and communications Security, June 2000, pp. 5-8.