The Monitor and Principals

By

Edwin A. Melendez

Contents

Dedication

My family provides me with inspiration and support. In part, I am who I am thanks to them. I am a lucky person to have such a loving family. My father, Edwin, is always there to give me advice and lend a helping hand. My mother, Carmen, never fails to show me the bright side of life and put a smile on my face. My brother, Jeffrey Melendez, is my role model. He impresses me and I look up to him for motivation. Finally my little brother and teen sister, Jonathan and Mariela live far away from me but I still keep them close to my heart.

Acknowledgments

I would like to thank Dr. Alec Yasinsac, my major professor, for all his effort in mentoring me. The weekly meetings and many discussions that we shared were necessary for the successful completion of this project. I am also grateful to the Systems group for their administration of the security lab. Lastly, I would like to thank my other two lab members, Sachin Goregaoker and Nikhil Patel, for their feedback and motivation.

Abstract

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

Security protocols are used for authentication and encryption key exchange. However, many of these protocols are flawed and vulnerable to attacks. The Secure Enclave Attack Detection System or SEADS examines the pattern of send and receives events in the execution of a protocol to detect intruders. This detection system consists of three major parts: the monitor, intrusion detection engine and knowledge base.

Since network messages may be encrypted, until now there was no known method of collecting information for security protocol intrusion detection. However, the SEADS's monitor uses a novel technique of gathering meta-information of network messages to detect attacks.

My master's project involves the creation of the monitor. In addition to this, I have developed a principal simulation environment used to test SEADS.

1. Introduction

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

An intrusion detection systems or IDS can add a level of security to a computer network by monitoring all the users and activity in its environment. Generally an IDS detects attack by analyzing the payload in messages or commands. However, in many cases this is not practical due to the fact that messages are often encrypted. Recently, a novel way of detecting intruders without looking at the contents of a message was introduced [1]. This system is called the Secure Enclave Attack Detection System or SEADS and it analyses meta-information about packets on the network. SEADS is applied specifically to the domain of security protocols.

Security protocols are commonly used on networks for authentication purposes and for the distribution of encryption keys. Since these protocols are used when secure channels of communication are established, they must be designed to be free from attack. Nevertheless, the literature shows that many of these protocols are vulnerable to intrusion by sophisticated intruders [4].

Normally, formal methods of verification are used whenever new security protocols are introduced. Nevertheless, formal methods have lacked the power to prove the absence of errors in protocols. Another limitation of formal methods is that they examine protocols offline, most likely in someone's research lab. As of now, there are no methods available to check a protocol online while it is being executed. These limitations make the implementation and use of security protocols vulnerable to attacks.

My master's project entails the development of a monitor that gathers events from encrypted sessions. The monitor uses a novel technique of gathering meta-information about network messages. Until now there existed no known method of collecting information for security protocol intrusion detection.

An integral element of my master's project was to develop a principal simulation environment. Principals are the names given to the processes that participate in the execution of security protocols. This simulated environment is used to test the functionality of the monitor by simulating normal, suspicious and attack behavior.

The rest of this paper is organized with the following objective in mind: present background information about intrusion detection systems, explain the inner workings of SEADS as described in [1] and finally provide a detailed description of the monitor and the principal simulation environment.

2. Background on Intrusion Detection Systems

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

The Common Intrusion Detection Framework or CIDF is a movement to develop ways to allow intrusion detection engines to interoperate with other programs [5]. One of their attempts is to architecturally divide the IDS into 4 major independent components that can be reused in other systems. These components consist of the following:

1. Event Generator

2. Event Analyzer
3. Event Database
4. Response Unit

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

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

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

```
...
su <correct password>
rm –R /*
```

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

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

3. Secure Enclave Attack Detection System (SEADS)

3.1 The Needham-Schroeder Protocol

To show how an intrusion detection program can detect an attack on a security protocol an example will be shown. The Needham-Schroeder Protocol or NSP is a popular and widely used key distribution and authentication protocol. This protocol was first introduced in [12] in 1978 and now countless papers show how intruders can spoof the participants by replaying messages. Nevertheless, this protocol is still widely used. The protocol contains the following messages:

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

This protocol achieves two-way authentication between participants A and B plus A and B are now in the possession of a common private key, which can be used to create a secure connection between the two. The protocol uses public-key encryption and the assistance of a central authentication server. In the first step of the protocol, A tells the authentication server S that he wishes to communicate with B. In return, the server replies with a message encrypted with A's public-key, kas.

{na,B,kab,{kab,A}kbs}kas

Inside the encrypted message lie the following items:

1. na – a generated random number called a nonce. The nonce is used to show the freshness of subsequent messages.

2. B – B's identification name.
3. kab - A private key whom A and B will share after the protocol has concluded.
4. {kab,A}kbs – A message destined for B encrypted with B's public key.

On step three, A sends to B the encrypted message received from S. B in turn replies to A with a nonce, nb, encrypted with the new private key, kab. A then proofs to B that he has the new key by sending to B a modified nb encrypted with kab.

The NSPKP protocol consists of 5 messages and it involves the participation of three parties. Since the messages in this protocol are encrypted, the contents of the message cannot be used to detect attacks. We must use other type of information. For instance, every message in the NSPKP protocol is sent by one participant and received by another. This series of send and received events are valuable information that does not involve the decryption of messages. The NSPKP protocol is shown below as a series of send and receive events:

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

Notice that the number of events in the protocol is twice the number of messages. This is due to the obvious fact that every message has a sender and receiver.

3.2 Detecting an Attack on NSPKP

It is now time to show an attack on NSPKP and show how the intrusion is detected. This particular attack was conceived in [3] and is referred to as the Lowe's attack. It requires the intruder to intercept messages from one session, opening a second session and replaying the intercepted messages on the second session.

Recall the last three message of the NSPKP.

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

The attack on NSPKP is shown below:

1-1.	A -> M:	{kab,na}kbs
2-1.	**M -> B:**	**{kab,na}kbs**
2-2.	**B -> M:**	**{nb}kab**
1-2.	M -> A:	{nb}kab
1-3.	A -> M:	{nb-1}kab
2-3.	**M -> B:**	**{nb-1}kab**

Even though NSPKP encrypted the messages, the intruder, M, was still able to obtain authentication from both parties, A and B. The intruder does not have to decipher the payloads in order to perform this attack. Instead, the intruder relies on copying and replaying messages. In the example shown above, M intercepts a message from A that was destined for B. He then opens a new session with B using the message he obtained from A. Subsequently, B replies to M with its reply, which in turn M uses to reply back to A. This technique of copying and replying messages continues in steps 1-3 and 2-3.

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

The attack on the NSPKP protocol shown above, can be describe with the following series of send and receive events:

Session 1	Session 2
1. A -> S	1. ---
2. S <- A	2. ---
3. S -> A	3. ---
4. A <- S	4. ---
5. A -> B	5. ---
6. ---	6. B <- A
7. ---	7. B -> A
8. A <- B	**8. ---**
9. A -> B	**9. ---**
10. ---	10. B <- A

The absence of events 1-5 in session 2 is not necessarily an attack signature since A can open multiple sessions with B using the reply it obtained from the authentication server, S, on session 1. However, the missing events 6,7 and 10 in session 1 and events 8 and 9 in session 2 are enough to declared the presence of an intruder.

3.3 The Topology of SEADS

SEADS assumes secure communication between the monitor and principals. Inside this protected environment, the principals can safely forward information to

SEADS. However, the principals communicate between one another via public networks such as the Internet.

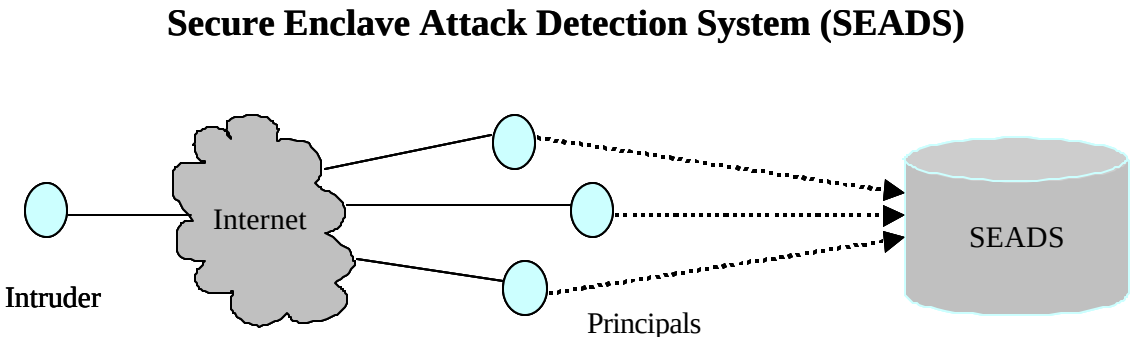


Figure 3.3

According to this topology, the intruder only interacts with the principals on the public network.

Since SEADS is such a large program, a divide and conquer approach was used in its design. The program was conceptually divided into three parts similar to the ones described in the CIDF model presented in section 2. The three parts are:

1. The Monitor
2. The Intrusion Detection Engine (IDE)
3. The Knowledge Base (KB)

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

3.4 Creating SEADS

The creation of SEADS began in a software engineering class taught by Dr. Alec Yasinsac in the fall of 2000. In this class, the project requirements were gathered, a preliminary design constructed and a crude prototype was developed. Three students, Sachin Goregaoker, Nikhil Patel and I continued the work on SEADS in the spring and summer of 2001. During this time, almost all of what is now SEADS was redesigned and recoded because the software produced in the software engineering class was not expandable. There were a few reasons for this. The main two reasons were: 1) our inexperience in programming for the Win32 platform and 2) the time constraint of developing a complicated piece of software in one class.

The project was divided into three parts. Sachin and Nikhil were responsible for creating the intrusion detection engine and knowledge base respectively. I was involved in the development of the monitor. In addition to this, I developed the Principal Simulation Environment.

The software that was developed runs on the Windows 2000 platform. It was written in C++ on the MS Visual C++ compiler. The code also uses the Win32 API, the Microsoft Foundation Classes (MFC) and the Standard Template Library (STL).

4. The Monitor

4.1 The Monitor's Database

In an intrusion detection system, the monitor is the component that gathers network traffic between principals. It packages this information into objects and stores them in the monitor's internal database.

The monitor is novel because it gathers information without looking at the contents of the network traffic. The information that is collected is meta-information about the traffic. In order to accomplish this, the principals are required to report events to the monitor. You may recall from section 3, that during the execution of a protocol, a series of messages are exchanged between principals. Each message in the protocol consist of two events: a send and a receive event. These send and receive events are the ones forwarded to the monitor.

Each time a principal reports to the monitor it packages the event in the following format:

[PN][Parties][Nonce][Event]

Each field in the packet has a purpose. Their description is listed below:

1. PN – the name of the protocol executed by this principal in this session
2. Parties – this is a list of all the principals participating in the session
3. Nonce – this number is used to differentiate between sessions among the same group of principals
4. Event – a send or receive event. This information also contains the two principals involved. For instance, A->B or A<-B.

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

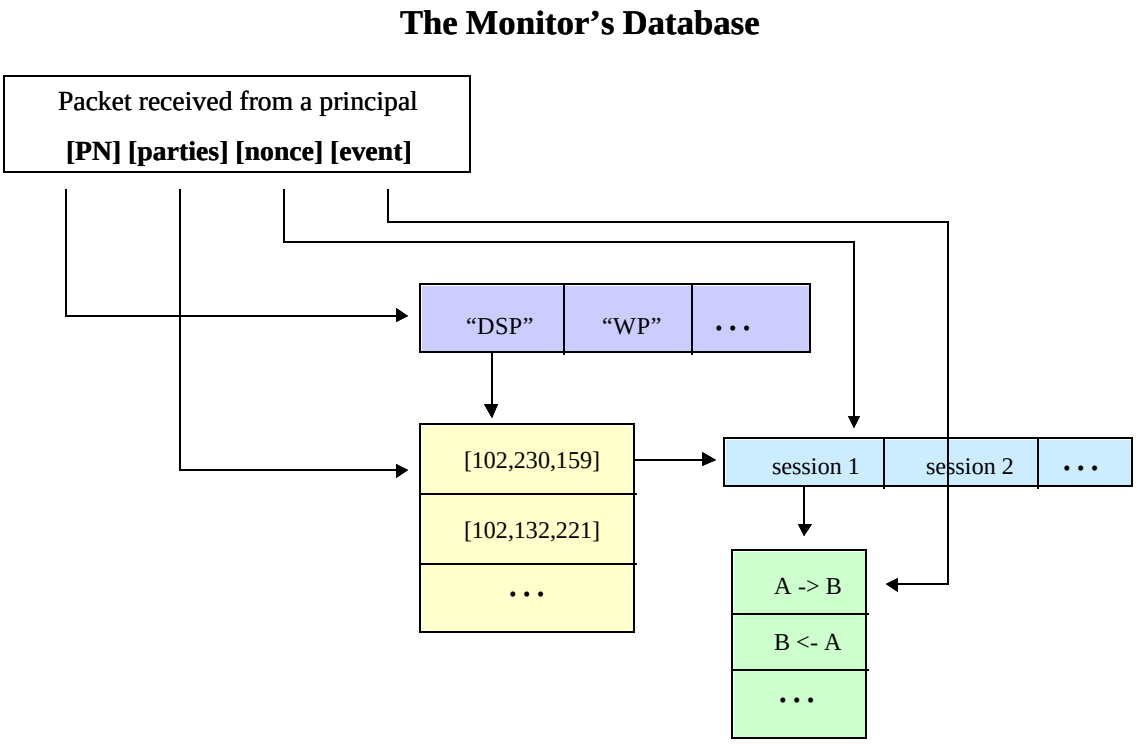


Figure 4.1

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

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

4.2 The Monitor's Threads

The monitor was designed to be robust and able to handle a multitude of sessions. One possible scenario for the monitor is handling hundreds of sessions concurrently. To accomplish this, a multi-threaded design was chosen for the monitor. This design is commonly referred to in the literature as the consumer-producer thread design. In this

case, the consumer is a thread, which is constantly listening to network socket connections and managing all the open sockets. Any information the consumer reads from a socket is quickly placed in a queue. The producer thread takes the packets waiting in the queue, checks them for proper format, converts them to event objects and then stores the objects in the monitor's database. After storing an object in the database, the producer thread signals the IDE engine about the presence of new events. In turn, the IDE uses a well-defined interface provided to it by the monitor to retrieve events from the monitor's database. This consumer-producer thread design helps the monitor handle many concurrent sessions by shifting the bottleneck from the network socket's I/O and into the internal, dynamic queue of the monitor. Figure 4.2 illustrates the monitor's threads collaborating in a consumer-producer thread design.

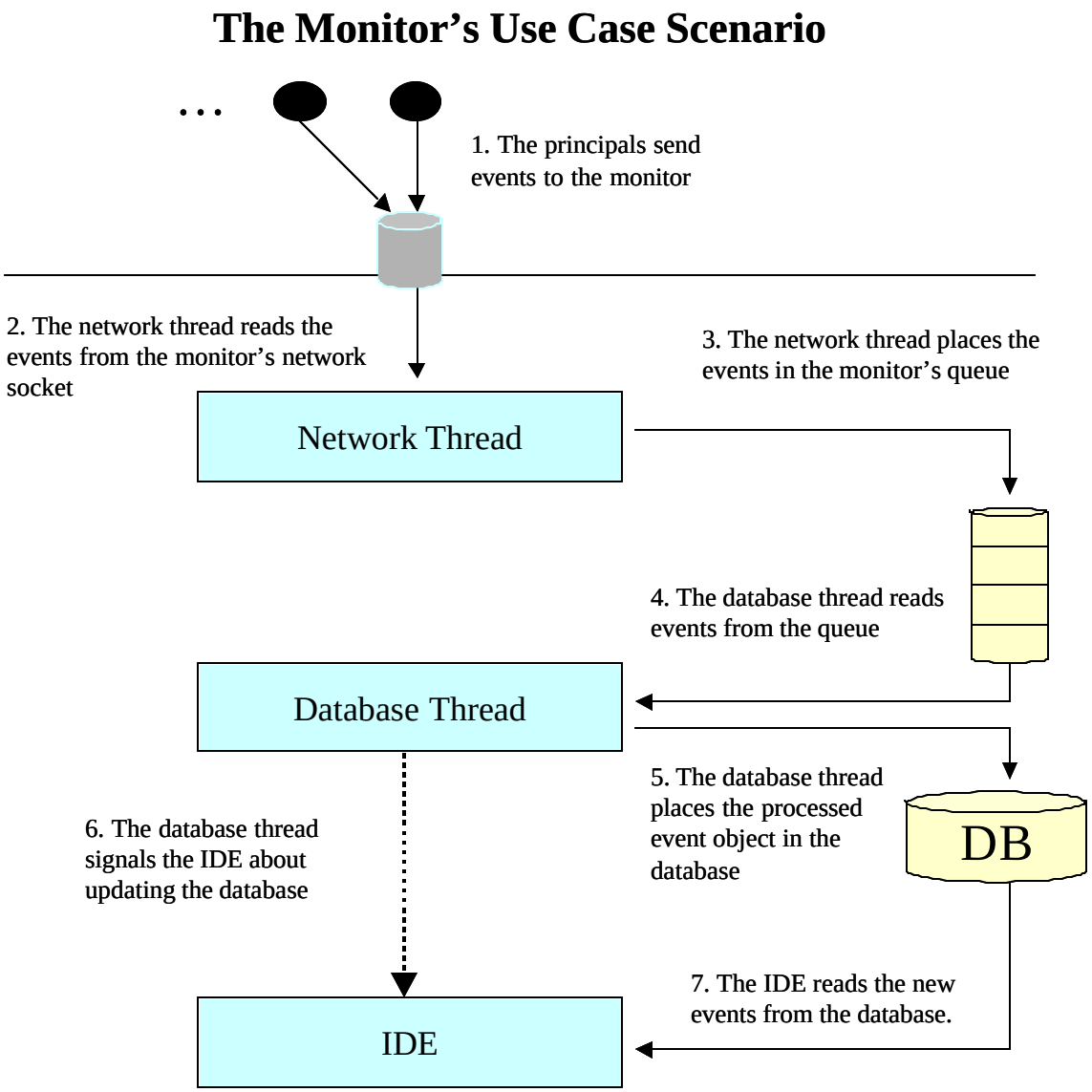


Figure 4.2

4.3 The Monitor's Code

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

signaling and synchronizing and the critical sections were used to assure the integrity of the monitor's data structures.

Standard Template Library containers were also used. For instance, the monitor's database was created with maps, linked-lists, and vectors. Since these containers grow dynamically, the monitor's database can hold as much data as possible limited only by the computer's memory. To learn more about the coding of the monitor please refer to the sections on the Appendix titled "Computer Skills Needed For The Project".

5. The Principal Simulation Environment

5.1 The Components of the Principal Simulation Environment

In order to test the functionality and correctness of SEADS, a network environment of principals was developed. These principals were created with the intelligence to initiate and engage in sessions involving other autonomous principals. The principals run on any Windows computer and they have the ability to execute any given protocol signatures over the network. As required, the principals always report the completion of events to the monitor. In addition, the principals have the ability to engage in different types of activity such as normal, suspicious or attack behavior.

The Principal Simulation Environment is a sizable system and is therefore divided into three different programs:

1. The Principal Simulator
2. The Principal Dispatcher
3. The Principals

The Principal Simulator provides the user-interface for creating an environment of principals. Each simulation requires the input of parameters that are used to configure the system. Some of the configuration parameters that the user can customize are the

number of sessions to be executed, the computers involved in the simulation and the protocols and signatures that the principals will execute. After the simulation has been created, it is the job of the Principal Dispatcher to instantiate the principals at a given computer when instructed to do so by the Principal Simulator. The autonomous principals then communicate with each other, execute protocols and report events to the monitor. The figure 5.1 illustrates how the three components interact to produce a simulation.

The Principal Simulation Environment's Use Case

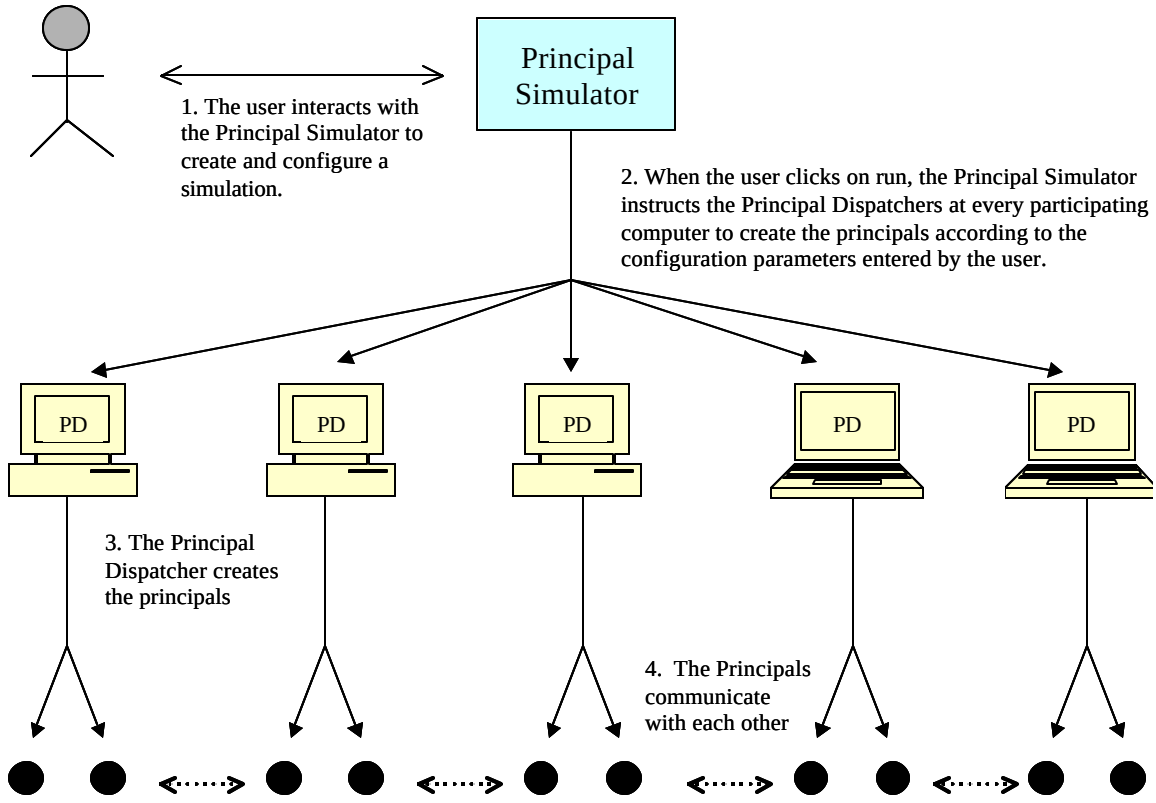
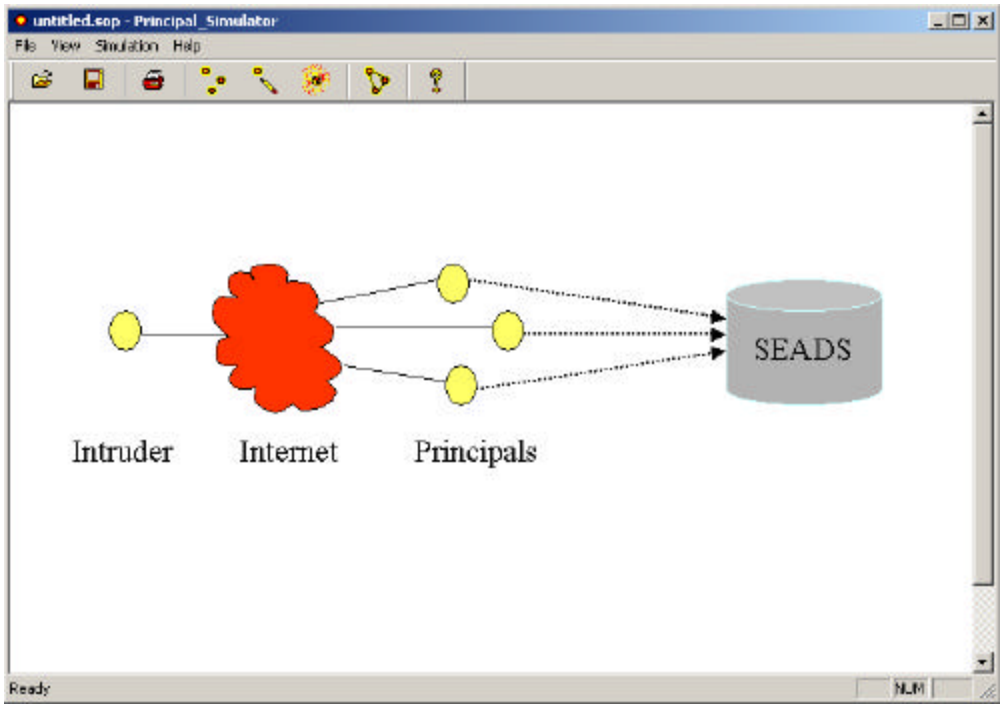


Figure 5.1

5.2 The Principal Simulator

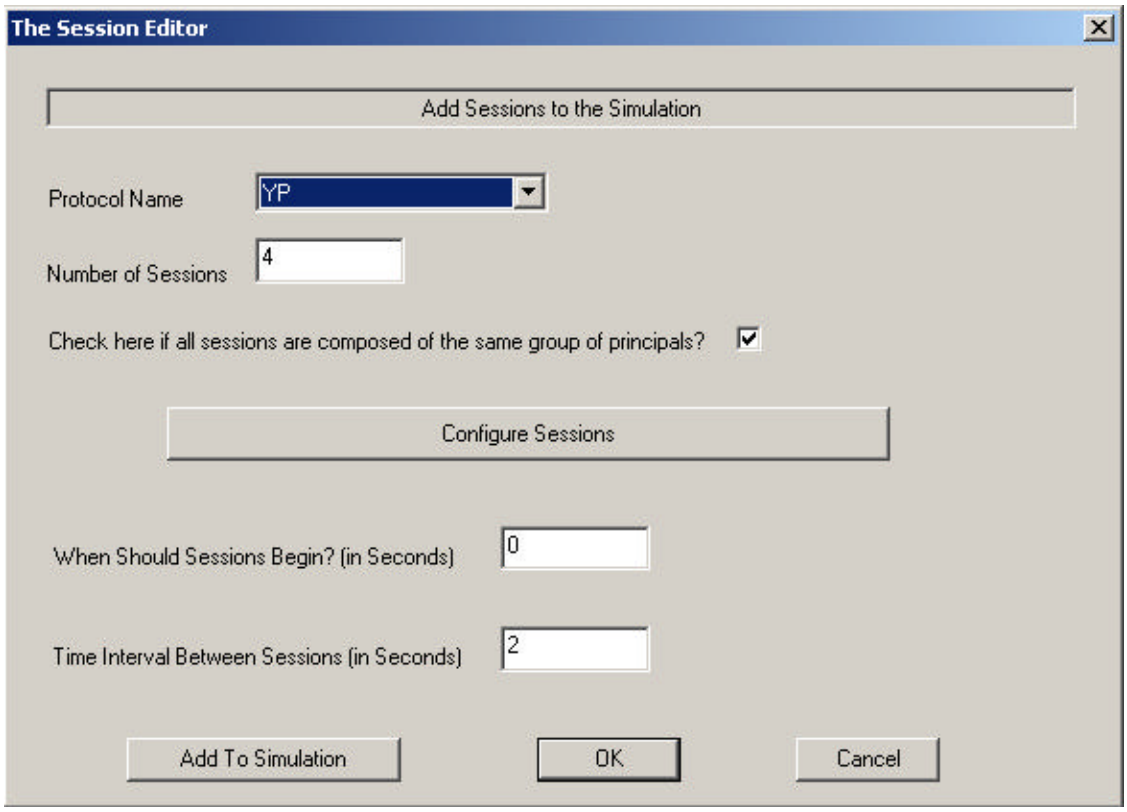
The Principal Simulator is the program that configures the network environment of principals. This is a GUI program that provides the user with an easy to use interface to create and run simulations.



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

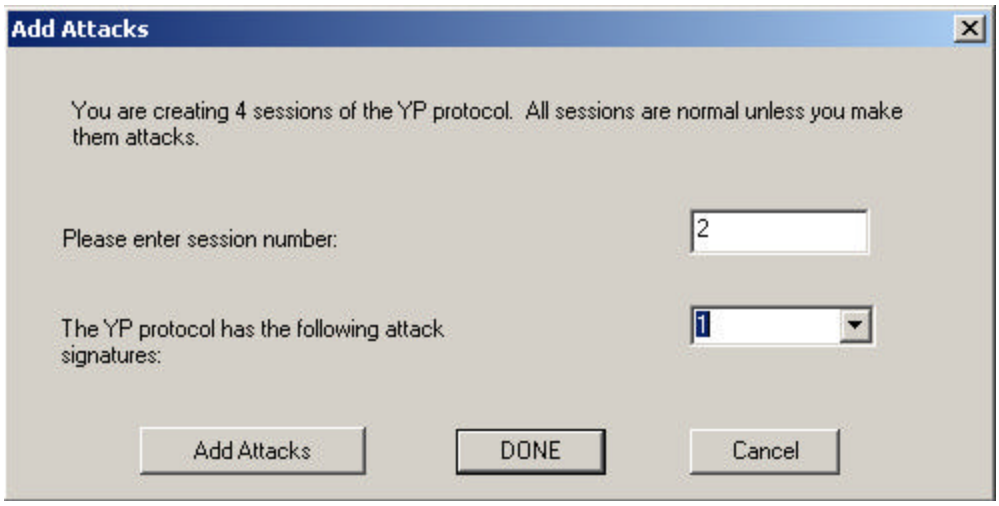
1. Protocol name
2. Number of Sessions.
3. Do all sessions consist of the same group of principals? (Check box)
4. Configure Sessions – What signature will the principals execute? (normal, suspicious or attack)

5. Start time

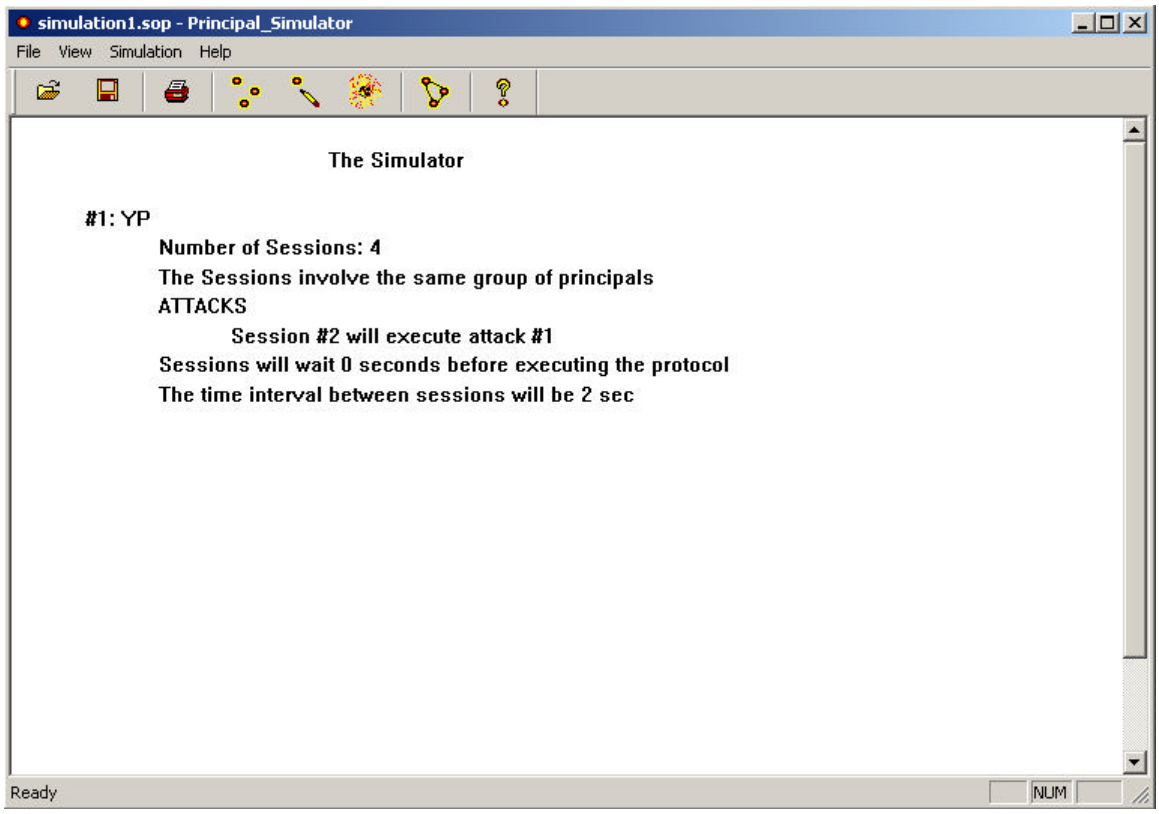


In order to assist the user in selecting a protocol name, the program reads the file containing the protocol signatures and populates the “Protocol Name” combo box with the available protocols.

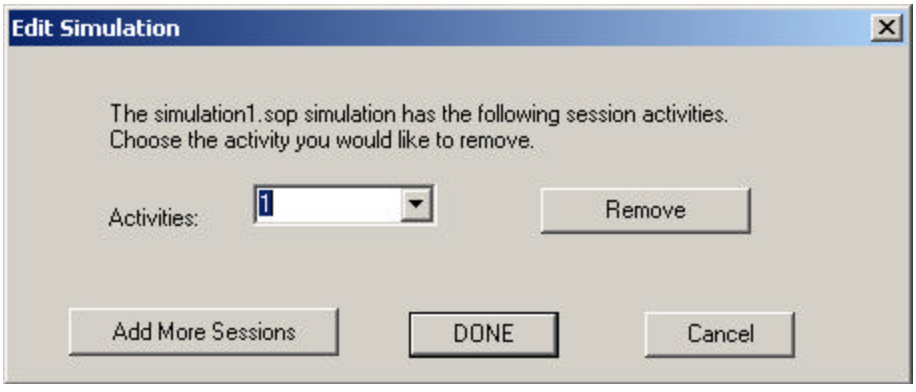
When the user clicks on the “Configure Sessions” button the following dialog box is displayed:



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



After the creation of a simulation, the user has the choice of editing, saving, printing or deleting the simulation. If he clicks on the edit button, a dialog box is shown that will allow the user to add or remove activities from the simulation.

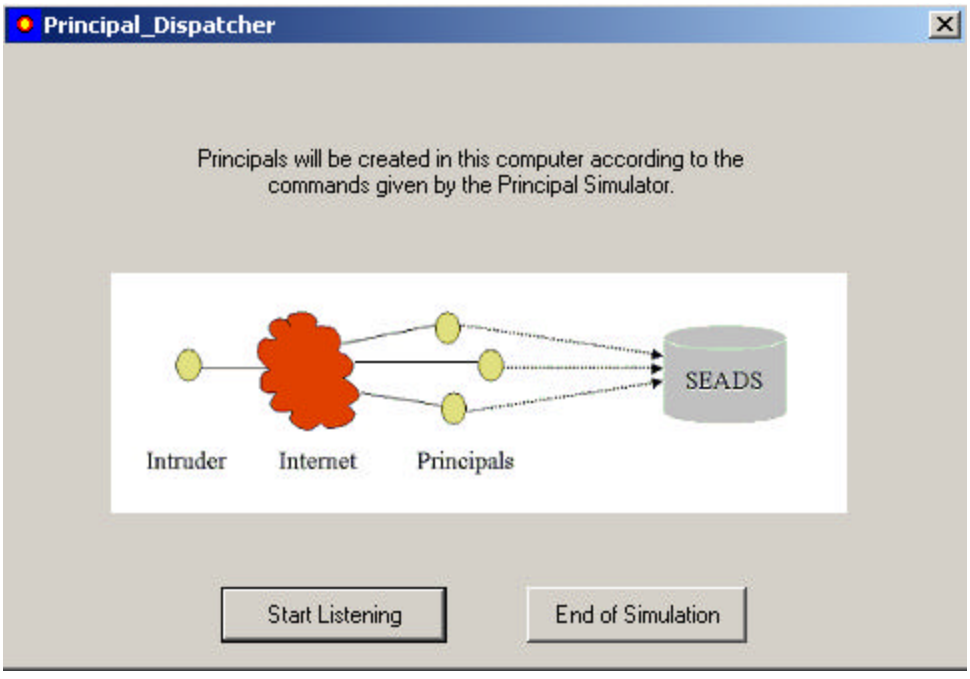


Once the user has finally loaded a simulation in the Principal Simulator he is ready to run it. When the user clicks on run, the Principal Simulator communicates with every Principal Dispatcher running on all the Participating computers. The Principal Dispatcher in turns creates the principals that will run on his computer.

5.3 The Principal Dispatcher

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

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



5.4 The Principals

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

This file, named “Simulation_File.txt”, is almost identical to the one provided to the intrusion detection engine by the knowledge base. The file is divided by protocols and each protocol contains at most one normal signature. Any additional signatures in the file represent suspicious or attack scenarios. An example of a signature present in the file is the one shown below executing a normal session of the Denning-Sacco Protocol or DSP:

```
begin DSP -1
A -> S
S <- A
S -> A
A <- S
A -> B
B <- A
end
```

When the Principal Dispatcher creates the Principal the first step is to read “Simulaiton_File.txt”. After picking the selected signature from the database the Principal determines the number of other Principals involved. This number may differ

since different security protocols differ in the number of participating entities. However, all protocols involve at least two principals.

The next step is to determine the initiating principal. The initiator is responsible for creating a random number called a nonce. As you may recall, the monitor identifies sessions by using the protocol name, group of principals and nonce. The initiating principal is now ready to send the first message to the corresponding party. During the execution of the protocol signature the principals report the event to the monitor. Figure 5.4 shows the flow chart of the Principal program.

The Principal's Use Case

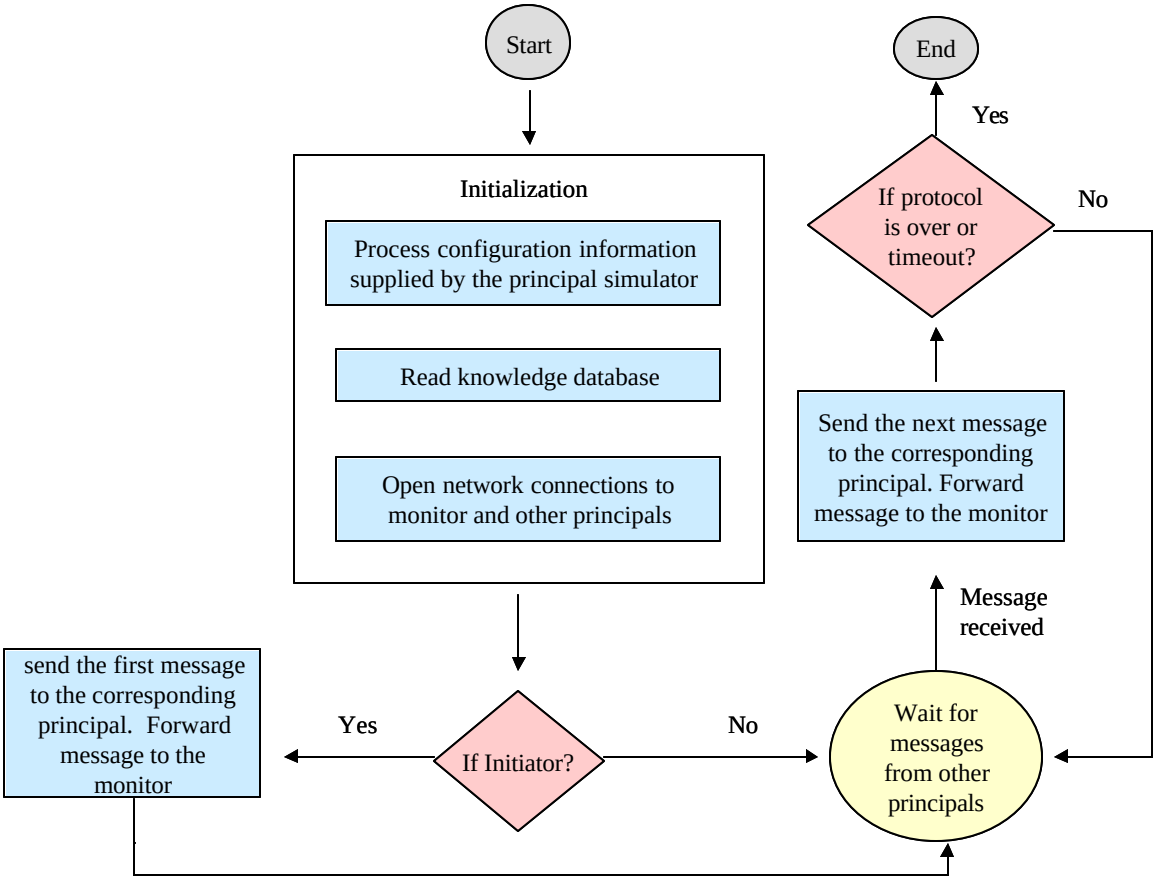


Figure 5.4

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

6. Test and Results

Extensive testing and demos were conducted to test the functionality of the Monitor and the Principal Simulation Environment. In all the tests, the software executed

according to specifications. In addition, the monitor was successfully integrated with the intrusion detection engine. In all the tests, SEADS detected the simulated suspicious and attack behavior.

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

Some of the most significant tests are listed on the Appendix under "Test and Results".

7. Conclusion

Why implement the monitor? How can the creation of this software contribute to security research? This monitor program shows that relevant and useful information can be gathered without having to examine the payload of messages exchanged between principals. This is very significant since encryption is becoming increasingly popular and will be widely used by computers in the future.

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

If future work were to be done, what kind of improvements would the monitor obtain? One thing that would be useful is to recreate the interface of the monitor according to the specifications listed by the Common Intrusion Detection Framework. In

this way, the monitor could theoretically be reused in other intrusion detection systems looking for a Windows monitor program. As you may recall, CIDF is a movement to divide intrusion detection systems into portable parts that can be reuse in other systems. CIDF was discussed in section 2. Another possible modification to the monitor would be to make it platform independent and mobile. In this way, the monitor could be deployed in different systems running a variety of operating systems.

Appendix

1. Test and Results

All tests were conducted in the newly created Computer Science Security Lab called SAIT. During the tests, six workstations running Windows 2000 were used. Two of the workstations were SUN Microsystems running Windows 2000 via a SUNPCI card. All the PCs were equipped with a Pentium III processor and 128MB of RAM. The tests were created using the Principal Simulator program presented in this paper. After the tests were created they were saved to files with .sop extensions. These tests are listed below:

Test # 1 – "StressTest_5.sop"

The file "StressTest_5.sop" contains a compilation of 11 sessions divided into 5 activities. The session parameters are listed in the following table:

Activity # and Protocol Name	**Session #**	**Signature**	**Group of Principals**	**Starting Time**	**Total # of Principals**
1. DSP	1	Attack #0	Different Groups	0 seconds	3*3 = 9
	2	Normal		2 seconds	
	3	Normal		4 seconds	
2. DSP	1	Attack #0	Same Group	0 seconds	3
	2	Attack #0		2 seconds	
3. ISKOPUAP	1	Attack #0	Different Groups	4 seconds	2*2 = 4
	2	Attack #0		6 seconds	
4. ISKTPUAP	1	Attack #0	Same Group	5 seconds	2
	2	Attack #0		7 seconds	
5. ORP	1	Attack #0	Different Group	8 seconds	2*3 = 6
	2	Attack #1		10 seconds	
Total # of Principals					24
Status of Test					**Success**

Note that the number of principals involved in each session is determined by the protocol being executed. For instance, the DSP protocol requires the participation of three principals.

The main objective of this test was to examine the correctness of the simulator. Could the Principal Simulation Environment create principals that executed according to the parameters entered by the user? Specifically, the principal's starting time and its attack signature.

Test # 2 – "ParallelStress.sop"

The file "ParallelStress.sop" contains a compilation of 8 sessions divided into 4 activities. The session parameters are listed in the following table:

Activity # and Protocol Name	**Session #**	**Signature**	**Group of Principals**	**Starting Time**	**Total # of Principals**
1. WLAPF	1	Attack #0	Same Group	0 seconds	3
	2	Normal		3 seconds	
2. WLAPF	1	Attack #0	Same Group	1 seconds	3
	2	Normal		13 seconds	
3. ORP	1	Normal	Same Group	2 seconds	3
	2	Attack #1		4 seconds	
4. ORP	1	Normal	Same Group	4 seconds	3
	2	Attack #1		16 seconds	
Total # of Principals					12
Status of Test					**Success**

The main objective of this test was to show that the Principal Simulation Environment is capable of simulating parallel and replay session attacks. These types of attacks involve the same group of principals engaging in multiple sessions. In parallel session attacks, the first session starts and then blocks for a predefined amount of time. While the first session is blocking, the second session executes. Once the second session is finished, the first session unblocks and continues executing until it finishes. Replay session attacks are similar to parallel session attacks except that the two sessions do not interleave. Recall

that information about any given attack is available on the signature file. The principal's job is to read the proper signature and execute it accordingly. This test showed that the principals were able to execute complicated signatures available on the file "Simulation_File.txt".

Test # 3 – Combination

This test consisted of running the monitor for one hour and seeing if it could handle a magnitude of sessions. In this test, several test files were executed repeatedly. The "Other Files" listed below are test files that are variations of the StressTest_5.sop and ParallelStress.sop files.

Test File	Number Of Times Executed	Total # of Principals
1. StressTest_5.sop	> 25	> 24 * 5
2. ParallelStress.sop	> 10	> 12 * 5
3. Other Files	> 10	> #P * 10
Status of Test		**Success**

This test showed that the monitor is robust, stable and accurate. It handled a great deal of information and operated for a prolonged period of time. The large number of sessions generated a great deal of network traffic, which the monitor had to store in its database. After the test, the monitor's database was analyzed and as hoped, there was no loss of information. In addition, all the events were accurately organized in the database.

2. Computer Skills Needed For The Project

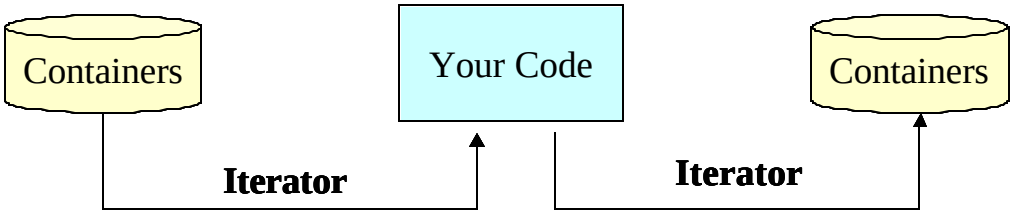
The Monitor and the Principal Simulation Environment are network and multithreaded software that runs on the Windows platform. They use complex data structures and algorithms. Moreover, the Principal Simulation Environment uses GUI programming for its user interface. In all, this was complicated software to write and it required the learning of new programming skills. The following is a list of skills used during the coding phase of this project:

I. **STL Containers** – The Standard Template Library containers are template classes that can contain literary any data type including other containers. The containers can grow dynamically as you add elements and their interface is standardized and easy to use. The containers used in my project are listed below:

a. Maps – these are associative arrays implemented as binary trees. They are also called dictionaries. They are convenient since they allow the programmer to index the array with different data types.
b. Linked-lists – these containers are useful if you require a data structure that can efficiently delete or add items anywhere on the list.
c. Vectors – useful if you need a dynamic array.
d. Queues and Strings

One of the fundamental designs of STL is based on the separation between data and operations. All the containers provide iterators with similar interfaces to read the containers. Therefore, if the programmer wishes to change the container type in the middle of the project the code changes little. This idea is shown pictorially below:

STL is based on a separation of data and operations



II. **Win32 Programming** – It was very rewarding and convenient programming for Windows thanks to the many tools and documentation available. First of all, credit has to be given to the Visual C++ compiler. It is superb. It really helps the programmer be more productive. Its debugger, color coded text, wizards, controls and pop-up combo-boxes and other features makes the life of the programmer a lot easier. The Visual C++ compiler also has the assistance of the MSDN library, which documents the Win32 API. The library contains hyperlinks that makes it easy to navigate the documentation.

The Win32 API was also convenient. Threads, network sockets, events and critical sections are all considered kernel objects. They are robust and well documented. Event objects were used as signal for reading sockets, and synchronizing threads. The critical section objects were used to maintain the integrity of the data structures when multiple accesses by different threads were a possibility.

The Microsoft Foundation Library or MFC was used to create the GUI of the program. MFC provides the programmer the ability to easily create user-interfaces with features that users have come to expect from software. One of these features is printing capability. In general, MFC allows the programmer to create programs that have the "look and feel" and streamline characteristics of popular Window programs.

One drawback about Win32 programming in general is its propriety nature. It refuses to be POSIX compliant. POSIX is an interface standard and it stands for Portable Operating System Interface. If a program is written with the POSIX interface when using sockets, threads and other kernel objects then the program can easily be ported to other computer platforms. However, Microsoft worst nightmare would be if Windows programs were easily ported to other computers. Therefore, it is very hard or nearly impossible to write program for windows that are platform independent.

Bibliography

[1] Alec Yasinsac, "Detecting Intrusions in Security Protocols", Proceedings of First Workshop on Intrusion Detection Systems, in the 7th ACM Conference on computer and Communication Security, June 2000

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

[3] Gavin Lowe, “Breaking and Fixing the Needham-Schroeder Public Key Protocol Using FDR”, In Proceedings of TACAS, Vol. 1055 of Lecture Notes in Computer Science, pp147-166, Springer-Verlag, 1996

[4] John Clark and Jeremy Jacob, “A Survey of Authentication Protocol Literature: Version 1.0”, 1997

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

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

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

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

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

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

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

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