


2007

The intrusion collector and emulator

Matthew Kyle Wilden
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Computer Sciences Commons](#), and the [Electrical and Electronics Commons](#)

Recommended Citation

Wilden, Matthew Kyle, "The intrusion collector and emulator" (2007). *Retrospective Theses and Dissertations*. 14540.
<https://lib.dr.iastate.edu/rtd/14540>

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

The Intrusion Collector and Emulator

by

Matthew Kyle Wilden

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Co-Majors: Information Assurance, Computer Engineering

Program of Study Committee:
Douglas Jacobson, Major Professor
Thomas Daniels
Jennifer Davidson

Iowa State University

Ames, Iowa

2007

Copyright © Matthew Kyle Wilden, 2007. All rights reserved.

UMI Number: 1443073

UMI[®]

UMI Microform 1443073

Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF FIGURES	iv
ABSTRACT	v
CHAPTER 1. INTRODUCTION	1
1.1 ISEAGE Tools	1
1.2 Replaying Traffic	2
1.3 Initial ICE Components	3
1.4 Paper Structure	5
CHAPTER 2. SOFTWARE BACKGROUND	6
2.1 Snort	6
2.2 MySQL	8
2.3 Libnet	9
2.4 Libpcap	9
CHAPTER 3. RELATED WORK	11
3.1 TCPReplay	11
3.2 RolePlayer	12
CHAPTER 4. DESIGN AND IMPLEMENTATION	14
4.1 Stages of Development	14
4.2 Stage One	15
4.2.1 Snort	15
4.2.2 MySQL	16
4.2.3 Replayer	17
4.2.3.1 Command Line Options and SQL Queries	19
4.2.3.2 Packet Creation	21
4.2.3.3 Packet Writing	25
4.2.3.4 IP Header Retrieval	26
4.3 Stage Two	27
4.3.1 Snort	28
4.3.1.1 Barnyard	28
4.3.2 MySQL	30
4.3.3 Replayer	31
4.3.3.1 Libnet and SQL Debugging	31
4.3.3.2 Presentation to User	33
4.4 Stage Three	36
4.4.1 Replayer	36
4.4.1.1 Code Cleanup	39
4.4.1.2 MySQL Configuration File	40
4.4.1.3 TCP Three-Way Handshake	42
CHAPTER 5. TESTING	47
5.1 Testing Methodology	47
5.2 Test Environment	48

5.3 Test Execution	50
Chapter 6 Future Work	53
6.1 Graphical User Interface	53
6.2 Monitoring Sequence Numbers	54
6.3 MySQL Database Indexing.....	54
CHAPTER 7 CONCLUSION	56
APPENDIX A. MYSQL TABLES.....	57
APPENDIX B. SQL Code	58
APPENDIX C. GLOBAL VARIABLES AND STRUCTURES	60
APPENDIX D. EXAMPLE MYSQL CONFIGURATION FILE.....	62
BIBLIOGRAPHY.....	63

LIST OF FIGURES

Figure 1. ISEAGE Tools and Environment	2
Figure 2. Initial Design Modules	18
Figure 3. Two IP Header Retrieval Design Options	21
Figure 4. Libnet TCP function prototype.....	23
Figure 5. Libnet IP build function prototype	25
Figure 6. Initial and Revamped Option Retrieval and Database Connection Designs	42
Figure 7. Test Environment	49

ABSTRACT

The ISEAGE environment is an advanced test-bed designed to allow researchers to play out real attacks against real machines without endangering any outside networks. Along with this test-bed comes the need for tools to utilize the environment and to help advance and grow the overall ISEAGE system. An important tool to have in any such test-bed environment such as ISEAGE is the ability to replay traffic from previous sessions. More specifically it is important to have the ability to replay attack traffic. It is this need that created the Intrusion Collector and Emulator (ICE).

ICE is a system comprised of three main components; Snort, MySQL, and a custom piece of software called the Replayer. These three pieces come together to form a cohesive unit that allows users to capture and store attack traffic for later study and use. The Replayer can then retrieve any and all of the attacks and replay them. This gives researchers using ISEAGE a valuable tool that will allow them to capture attacks from various real world sites using Snort, and then study their effects on machines and networks as they replay those same attacks in safety on the ISEAGE network.

CHAPTER 1. INTRODUCTION

The research being done in the fields of computer and network security is expanding more and more in this day and age. Along with this increase in research comes new facilities; one of them being the Internet Scale Event and Attack Generation Environment (ISEAGE) [1]. ISEAGE is a unique research test-bed located at Iowa State University that allows for real attacks to be carried out against real computers in a controlled area that mimics the Internet. Similar to any new test-bed of this kind, in order to take full advantage of the ISEAGE environment, new tools and devices need to be created.

1.1 ISEAGE Tools

Several tools have already been created and put into use in the ISEAGE environment. The most notable tool is the Traffic Mapper created by Dr. Doug Jacobson. It is the backbone of ISEAGE and gives it the ability to route and handle the network traffic running on the ISEAGE network. A simple traffic generation tool created by Noah Korba has also been put into use to help manufacture background traffic during various ISEAGE events. As on the real Internet, there is always an abundance of traffic that does not belong to each user and the traffic generation tool helps to emulate that on ISEAGE. Another tool also in use at the ISEAGE facility is the visualization software written by Nate Karstens. This software is used primarily during Cyber Defense Competitions (CDC) [2] and helps the CDC participants and visitors see what type of traffic is currently on the network and the most recent scores. Figure 1 shows some of the tools in use and in development for ISEAGE.

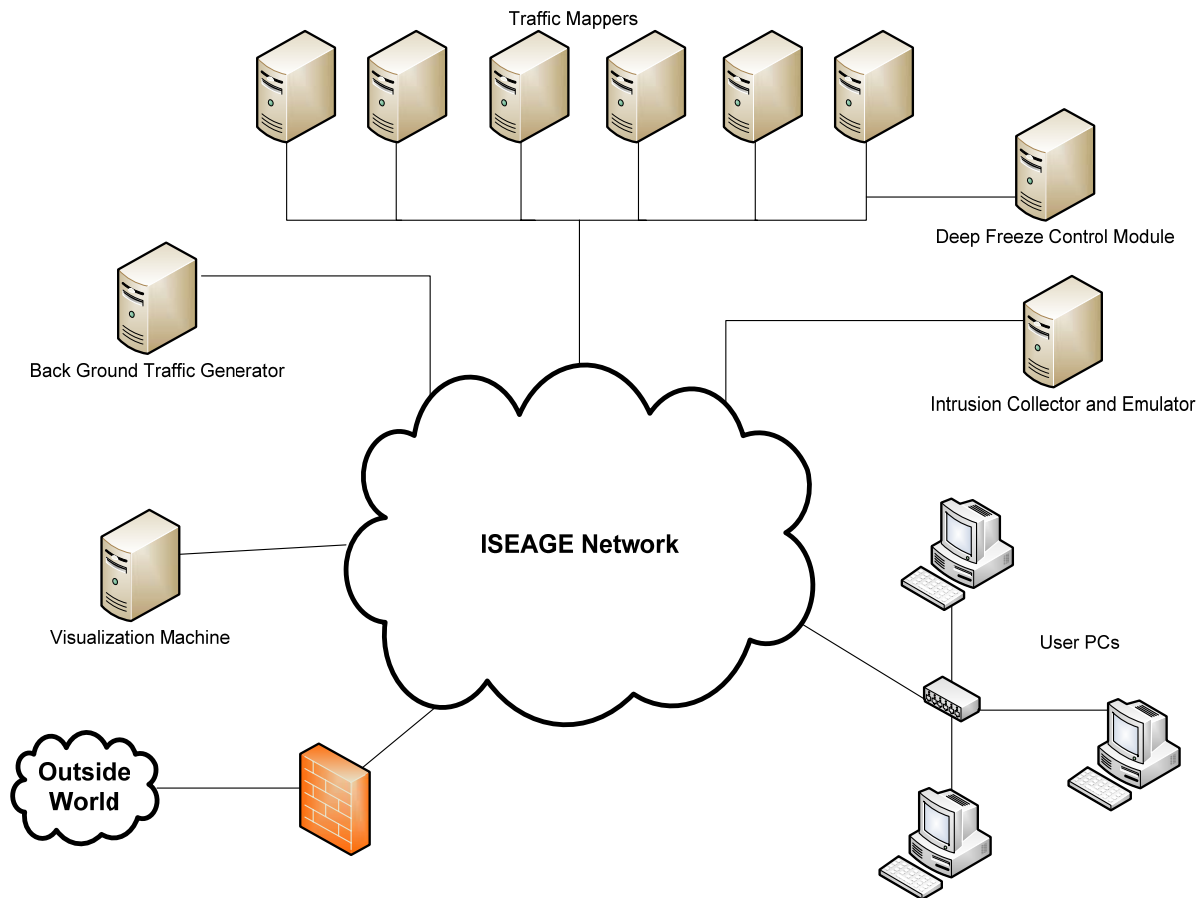


Figure 1. ISEAGE Tools and Environment

These tools were developed out of necessity for ISEAGE and there are many other tools that are needed as well to give ISEAGE the diversity it requires to function as an advanced test-bed. One tool required is the ability to replay attacks that have been captured in previous traffic sessions from either the ISEAGE facility or a corporate network.

1.2 Replaying Traffic

Replaying all types of network traffic is very important in any test-bed environment similar to ISEAGE as it gives the researchers the ability to study different parts of the traffic stream for varying purposes. These purposes can range from debugging software, to setting

up routers, to testing new services, or many other different needs that may arise and require traffic playback. For ISEAGE, creating quick background traffic on the network or allowing previous events held on the ISEAGE network to be studied for future events, are just a few of the uses gained by replaying network traffic. When it comes to replaying attacks, however, a specific tool is required apart from a standard traffic replayer.

While replaying traffic gives researchers at ISEAGE the aforementioned abilities, they also need the ability to study specific attacks and this can not be accomplished by replaying an entire traffic session. Replaying only attack traffic allows researchers to focus on the packets associated with each attack and study exactly what happens with each machine involved. At ISEAGE replaying attacks can also be used as a learning tool. Replaying attacks back to CDC participants after the competition is just one example of the many uses an attack replayer would have.

It is the need for a tool that can replay attack traffic that created the Intrusion Collector and Emulator (ICE) project. The goal of ICE is to take the data output from an intrusion detection system (IDS), store the resulting attack information into some type of database, and have a tool that allows the user to select from the stored attacks and replay them onto the ISEAGE network.

1.3 Initial ICE Components

Monitoring traffic for attacks and replaying network traffic are not new ideas. There are a variety of tools online already that perform both of these functions well. One such monitor is the IDS known as Snort [3] and was considered from the beginning to be used for the ICE project. It is readily available online and has an excellent track record as a reliable

IDS. The initial decision to use Snort came from the suggestion of Dr. Doug Jacobson. As the lead director of the ISEAGE project, Dr. Jacobson was in the position to make sure that ICE used components that he thought would benefit the project the most. As such, he also recommended using MySQL [4] as the database software to store the attacks with.

The first tool looked at to replay traffic was Tcpreplay [5]. At first glance, Tcpreplay looked to fit the needs of the ICE project. It has the ability to replay libpcap [6] dump files and Snort does have the ability to output data in a pcap format. However, it lacks a critical function required by ICE. Tcpreplay takes the whole dump file and will replay specific IP addresses the user wants to replay or can replay the entire file. There is no in depth customization available to the user to let them see and choose the individual attacks that they wish to replay. This is a key requirement of the ICE project. Tcpreplay is a great traffic replaying tool but is not an excellent attack replayer for ICE.

After more research it became obvious that the problem of replaying attacks from a Snort-created database was just a small issue in the overall field of replaying tools. It also became apparent why there was no one replayer that could do everything everyone wanted. Unlike Tcpcap which can grab network traffic in any number of ways and can be suitable for many different specific monitoring tasks, replaying tools have not reached that level. To understand why, it is necessary to recognize that network attacks will vary tremendously from one another and how they are stored will vary as well. This leads to the problem of trying to replicate those attacks in an all in one tool. Needless to say, it is a very daunting task to achieve.

What has happened due to this is the development of many different types of attack replayer tools that all fulfill different needs. Each has their own use and specific

requirements to run correctly. The ICE project required Snort to send its output into a database so that it would be easy to read through and be easily retrievable for an attack replayer. Due to this very specific requirement set, there were no replaying tools yet developed to handle the problem. Thus, a custom built replayer was necessary and the ICE project began to take shape.

1.4 Paper Structure

The rest of this paper will be structured as follows. Chapter 2 will detail the background of Snort and the other software components used as part of the ICE project. Related tools that have been designed for replaying traffic and attacks will be looked at in Chapter 3. The design and implementation of the Replayer will be covered in Chapter 4. Chapter 5 will go over the testing and evaluation of the Replayer. This will be followed up with future work to be done in Chapter 6 and a final conclusion in Chapter 7.

CHAPTER 2. SOFTWARE BACKGROUND

In order to fully understand the ICE project and its ultimate goal, it is first necessary to give a brief explanation of the organizational makeup of ICE. There are three major components that make up the complete ICE system. The first component is the intrusion detection system Snort. The second component is the database run by MySQL software which stores the output given from Snort. The final section of ICE is the piece of software known as the Replayer, which takes the attack information from the MySQL database and replays it. This chapter will cover Snort and MySQL, discussing primarily their background and why they were selected to be used in the ICE project. Also to be looked at in this section is Libnet [7], the library of functions for the C language that was used to create the Replayer, and libpcap, another library used to capture packets from a network device.

2.1 Snort

The Snort IDS was first developed in 1998 by Martin Roesch [8]. The initial intent of Martin was to develop a simple software package that could detect intrusions. From that humble beginning though, Snort has come to have an enormous presence in the security sector and is regarded as a solid and highly advanced IDS package. It was chosen to be used for the ICE project due to its large following and support online as well as its ease of use and extensive rules package which will be discussed below.

Snort has the ability to run as a simple traffic sniffer, a packet logger, and a network intrusion detection system (NIDS). For the purposes of the ICE project, Snort would be used exclusively as a NIDS. There are at least two approaches that can be used in implementing

an NIDS. One such method is a heuristic-based NIDS. This type of NIDS will have a record of what normal traffic patterns look like for the network it is monitoring. When a different traffic pattern would present itself on the network, the NIDS would send out an alarm to let the administrators know that there is an anomaly in the traffic.

The Snort NIDS works in a slightly different manner. It is based on the second type of NIDS which is rule-set detection. Snort operates by using a large library of rules that contain information on known attacks. Snort then takes its rule-set and checks each packet on the network against the rules. If a match occurs, then Snort will alert the administrator and log that packet to its output database for observation later. Typically, each rule is written for one type of attack. This allows the administrator to customize his or her rule-set to watch for the type of attack traffic they're most likely to get at their location.

Checking one packet at a time worked well for Snort at the beginning of its inception. Later on during its lifecycle in the years of 2001 and 2002 the preprocessor known as stream4 was developed. When Snort operates, it takes each packet through a variety of stages when it's checking them for matches against a rule. In order to allow Snort to do this job more efficiently, preprocessors were developed that would run various jobs on packets before they were put into the rule checks. In this way, it would leave more of Snort's main processing power available to check the rules rather than running various maintenance tasks on packets. The stream4 preprocessor was especially important as it helped to maintain state in a TCP stream of packets. It also allows Snort to reassemble an entire TCP stream and check the final payload of the stream against rules rather than checking each packet individually. This enabled Snort to detect attacks that spanned multiple payloads. This

ability becomes of great importance to the creation of the Replayer in the later stages of its development.

Snort also has also gained the ability to log the information of each offending packet over the years. Currently Snort can take the information from any packet that it detects as a threat and store that packet's information into any number of different formats. As of this writing, it is able to output into a TCPdump style format; a CSV file; any number of databases including, but not limited to, MySQL and Oracle; its own unique Unified format; along with a variety of small alert log formats. The Unified format is an important element that is unique to Snort and will be discussed in more detail during the design and implementation phase in Chapter 4. As for the database outputs that are available to Snort, the only one of interest in the development of ICE was the MySQL output. This output plugin allows Snort to record all of the attack information and warnings along with the packet information itself into the MySQL database. This creates an extensive database that holds all of the attack information from a Snort session.

2.2 MySQL

MySQL is a popular relational database in use today with over 6 million installations. The SQL section of MySQL stands for Structured-Query-Language and the language used to operate the MySQL database is SQL. It was initially developed in May of 1995, and has gone on to become one of the largest established GPL [9] licensed databases in the world. For the ICE project it was selected to be used due to the native support that Snort has for it.

One of the great advantages of using MySQL is its ability to interface seamlessly with website applications. This was another advantageous reason for using MySQL; it gave me

the ability to monitor the database from a website rather than working exclusively on the command line with it. This would become extremely valuable as the database grew in size and complex with the amount of attack information stored.

2.3 Libnet

The Libnet packet creation library was developed by Mike D. Schiffman to allow for the easy and fast development of network related applications. Specifically, it is a set of libraries for the C language that enables the user to create almost all of the major protocols that are used with IP based protocols today. Libnet was designed with only the creation of packets taken into consideration. Unlike the many different socket programming libraries available, Libnet has no infrastructure available to receive packets. There for it is ill-suited to create a fully functional networking application all by itself. However when used with the libpcap library, which will be discussed below, it gives the user the ability to create any type of networking application they would need.

2.4 Libpcap

Libpcap is the name given to the pcap libraries that are used on Unix environments. Pcap itself is short for packet capture and has two different ports of it. The first being the aforementioned libpcap and the other being Winpcap which is used for Windows machines. The goal of libpcap, and pcap languages in general, is to provide a framework to capture packet traffic from the network. Many different open source tools use these libraries in their development including TCPdump and Snort. Libpcap was chosen to be used for an add-on module that was created late in the development cycle of the Replayer. This will be

discussed later on in Chapter 4 when the implementation and design aspects of the ICE project are covered.

CHAPTER 3. RELATED WORK

Before probing into the inner workings of the layout and design of the ICE project, it is important to acknowledge and report on the status of the other tools that have been created to address the issue of replaying attack traffic. This chapter will cover two specific tools that have been created and will also go over their benefits and why they were not suitable for ICE.

3.1 TCPReplay

TcpReplay has already been mentioned briefly in the introduction and will therefore be covered first. As was stated in Chapter 1, TcpReplay functions by taking the output from the program Tcpcap, which uses the libpcap libraries to capture traffic from the network. TcpReplay has many options available to its user and has undergone tremendous development in the last few years. It has turned into a stable replaying system and will continue to improve in the future.

Unfortunately it is the output from Snort and Tcpcap, and the fact that TcpReplay was in a very preliminary stage at the start of the ICE project, which hindered using it in the ICE project. Snort is capable of having its session information formatted into the same Tcpcap pcap format for output. This would permit for TcpReplay to then take that session information and replay it. However, it gives no options to the user on what attack to replay and which specific packets are linked to each attack. In order for a user to determine which packets were linked to each attack, they would have to manually go through the session output and match up each packet with its associated tag in the warning log created by Snort.

This could also be scripted if necessary but is an extremely long and tedious way to gather the information from a Snort session. That is why the output from Snort was sent to a MySQL database instead; it gives the user a much easier way of navigating and processing the information taken from Snort. By using the MySQL output on Snort, Tcpreplay was removed as an option, not for its lack of replaying functionality, but for its lack of feasibility in use with Snort sessions.

3.2 RolePlayer

The tool known as the RolePlayer [10] was developed at the University of California, Berkley by Weidong Cui, Vern Paxson, Nicholas C. Weaver, and Randy H. Katz. It is a relatively new development and is a very sophisticated tool. Where Tcpreplay and ICE's own Replayer are tools that will replay captured traffic, RolePlayer has not only the ability to replay attack traffic, but also to act as the host to incoming attack traffic.

A key difference between the Replayer and RolePlayer is the method in which each tool obtains the attack information they will be replaying. The Replayer receives the attack data to be replayed from the MySQL database, which in turn received its data from Snort. The RolePlayer, on the other hand, is trained to replay specific attacks. The RolePlayer is set on a network and has each attack that is to learn run by it. The RolePlayer then captures the traffic session and analyzes the traffic in order for it to replay the attack. This allows the RolePlayer the ability to learn very complex attacks but limits its library of attacks to replay to only the ones that it has been trained on and seen before.

During the training process there are also specific details that the RolePlayer looks for in each session. These details specifically deal with the values that will change when the

session is replayed. These can include such variables as time, cookies, a random key, or any other such item that might need to be altered when the attack is replayed. These variables are monitored carefully in order for the RolePlayer to maintain the state of each attack that it replays. When replaying an attack that has distinct parts of the packets that are time or sessions sensitive, the RolePlayer will modify those areas of the packet which will allow the attack to successfully occur. Again the RolePlayer must be trained for each of these attacks and the areas of the packets to look in for the dynamic variables.

The RolePlayer can also act as the host and client simultaneously if the user would like it to do so. The example given in [10] describes the RolePlayer receiving attack traffic and checking whether or not it is an attack that it has already been trained on. If it doesn't recognize the attack as a normal variant, it deems it worthy of further study. It then replays the attack traffic it received onto a host machine behind it on the network, in order for further testing of the attack to commence. Otherwise, the RolePlayer will simply drop an attack stream if it places it under an already known attack. It is this versatility and the ability to train and replay more advanced attacks that have made the RolePlayer a very unique and advanced replay tool.

It is the complexity and the way in which the RolePlayer receives its attack information that made it unsuitable for use with the ICE system. The goal of ICE is to correctly replay attacks detected and stored by Snort. With the RolePlayer designed to be trained in each attack it is to replay, it can not take the Snort output as an input. The RolePlayer also has more in depth functionality than what was called for the ICE system. That is not to say that the RolePlayer is inadequate, it is simply not the correct type of tool to use with ICE.

CHAPTER 4. DESIGN AND IMPLEMENTATION

The creation of the ICE project was a long and continuous process. This Chapter will focus on the different stages of development each component of ICE went through. The vast majority of this will be spent on the Replayer as most of the development time was dedicated to it and it was also undergoing constant revisions and enhancements during the implementation phase.

In order to organize this chapter into a coherent progression, each stage of development will be addressed. Within each stage, the major work and design of the components of ICE shall be gone over in detail. When that is accomplished, the next stage of development along with the various changes and design issues that occurred during that time period will be discussed. This will give the reader a chronological sense of how each piece went through development, and the challenges and successes that were overcome and achieved respectively, throughout the implementation of the ICE project.

4.1 Stages of Development

There were three distinct stages within the development of ICE. The first stage lasted four months and started January of 2005 and ended April of 2005. This stage occurred during the initial senior design team's work on the project. The second stage began in June of 2005 and lasted until August of 2006. During this stage, I continued work on ICE as well as other smaller projects that I had to attend to. The third and final stage of development lasted from August of 2006 to February of 2007. It was during this stage that final revisions

were put into place and some serious flaws within the Replayer component were discovered and corrected.

4.2 Stage One

In the fall of 2004, the senior design team consisting of myself, Shawn Rasmussen, Adam Hanson, and Li Yeow Chong was charged with developing what at the time was called an Attack Collector/Watcher/Replayer for ISEAGE. During the first semester of the senior design class, there was only paper work to be completed for the project. The actual implementation of the Attack Collector/Watcher/Replayer for ISEAGE did not begin until the spring of 2005 and at that time the project was also given the name ICE. The following sections will give a detailed overview of the initial design, implementation, and struggles that affected the three major components of ICE, Snort, MySQL, and the Replayer, during the first phase of development.

4.2.1 Snort

With Snort decided upon as the tool the Replayer would get the attack data from, it was necessary to study and research exactly how Snort operated so that it could be implemented properly and the correct data could be extracted from its alert logs.

With only four months available to the senior design team, the approach to Snort, and the other components as well, was simply 'get it working'. Getting Snort up and running turned out to be one of the easier tasks and took only a few days. However it would be several weeks while different options were looked at and selected in order to get a more sufficient output. Snort was installed on a Red Hat Fedora Core 3 Linux machine that was built for the ICE project by the senior design team. Installing was a minor task and the first

output from Snort was written directly to the screen. This was only done to test Snort and would not be continued in the future. By requiring Snort to write to the screen, it leaves a large chance that Snort will drop packets due to the processing time required for writing its output to window. In its place, Snort wrote to alert logs and the senior design team went to work studying those and the outputs that were given.

In each log there is enough information presented to the user to enable them to understand exactly what kind of attack occurred. There is not, however, enough to recreate a packet because there is no payload data represented in each alert entry. Also the extraction of the necessary information from the log to rebuild the attack would not be realistic. While it is feasible to break up a simple log entry, it would not be sufficient when dealing with hundreds of log entries. In short, it would be a naïve way to approach the problem. Therefore, the other outputs of Snort were examined. After consulting Dr. Jacobson, it was determined that a MySQL database would be the optimal solution for storing the information from the attack traffic. By using the MySQL output of Snort, we were able to grab the same information we would have gotten from the manually dissecting the log entries as well as the payload of each packet. It was a small task to modify the configuration file of Snort to produce the correct output to send to the MySQL database. There was no further research or development with the Snort software for the rest of stage one.

4.2.2 MySQL

When the decision to use MySQL was finalized, it became apparent rather quickly that no one within our senior design team had any experience with using the database software. Therefore, a large portion of the time spent with the MySQL database was taken to bring the group's knowledge up-to-date on the variety of versions available, as well as the

particulars of how to use the SQL language. The details of the SQL language and how we created the queries necessary with it will be discussed later in this paper, specifically in section 4.2.3. While looking over the SQL language, the group followed a variety of setup guides found on Snort's website, and was able to effectively setup the MySQL database in a relatively short amount of time.

Once the MySQL database software was running, a script was activated to populate it with a Snort database. This contained all of the tables and linking of those tables that was required to hold the information coming from Snort. The next step taken was to give the correct privileges to a user that would have access to the database. This user was then designated in the Snort configuration file in order for Snort to input the data into the database. This was the last step taken in dealing with the MySQL setup during senior design and no major changes came to the setup until stage two of development.

4.2.3 Replayer

Before looking into the design decisions that took place during the first stage of development, it is important to show the total design of the Replayer at that time. The diagram below in Figure 1 shows exactly what modules were planned for this first development stage and how each of them interacted to create the entire Replayer component. As can be seen, there were three main modules that were to be created; command line options module, the MySQL data module, and the packet creation and sending module. The purpose and design of each module will be covered below.

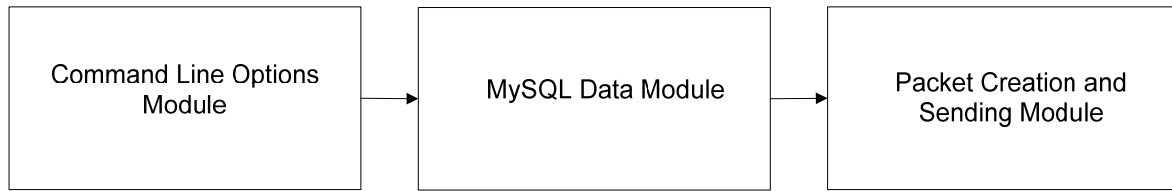


Figure 2. Initial Design Modules

Installing Fedora Core 3, setting up Snort, and setting up MySQL only took a little over two weeks to accomplish in the spring of 2005. The rest of the time was devoted to developing the Replayer. This occurred in stages with small development goals that would be met along the way. For the initial version of the Replayer that would be completed by the end of senior design, there were three main goals. The first goal was to have the Replayer receive command line options that would allow the user to select a port, a destination IP address, or a source IP address that would then be used to query the database. This would allow the user to select attacks based on the port used for the attack, the source IP of the attack, or the target IP address of the attack.

The second design goal was to implement SQL queries in the Replayer that were capable of grabbing the necessary information from the MySQL database. The figure in Appendix A shows a simplified version of which tables in the database are accessed for the various network headers.

The third and final design goal for the Replayer was to take the Libnet libraries and implement the packet creation and writing processes to facilitate the replaying of the attack packets. This was the largest and most complicated goal and would take most of the development time of the first stage.

4.2.3.1 Command Line Options and SQL Queries

The first step in the creation of the command line options module was to determine a way to retrieve the options from the command line itself in a quick and easy manner. After some initial research, our group discovered that a function known as `getopt` [11] was available in the C language. The `getopt` function takes the command line arguments given to the program as inputs. The function can then break down each argument and gives the user a way of implementing code based on what options are present. This allowed for quick and easy coding of a function that could take the output from the `getopt` function, create the necessary variables, and send those over to the SQL queries. This was accomplished in a matter of hours.

With the initial command line options completed, the attention of the design team turned to the SQL queries. The first necessity of this design goal called for the group to learn SQL. This was because the queries being used were more complicated than any of us had used before. The reason for such complicated queries arose from the knowledge that multiple tables would need to be searched and referenced against each other in order to obtain the correct packet data. As was shown in Appendix B, the entirety of the packet information was stored across various tables within the database. This in turn led to more complicated queries being constructed as more tables had to be accessed. The team took two weeks to go over books and study how the SQL language worked and what it took to design complicated queries. After much trial and error, a set of substandard queries were produced that were capable of grabbing the IP header information from the MySQL database. These queries would be optimized in the later development stages of ICE.

The queries that were in place at this point in time for the Replayer would successfully grab all of the IP headers that matched the port, the destination IP address, or the source IP address that was taken from the command line. When these entries were collected, they would be stored into a double array created by the MySQL library. One array kept track of which entry was being looked at, and the other array held the information from each field in the IP header. It was then that the first major design decision occurred. When the database was queried, it would return the all of the IP header entries that matched the query used. This process could take anywhere from half a second up to and beyond several minutes depending on how many IP header entries matched the query. It should also be noted that the MySQL database only contained in the area of 25,000 packet entries at this point. A database could grow much larger than that if Snort was left to run for an extended period of time.

With that in mind, the decision had to be made as to whether or not all of the IP headers, that matched the command line options given by the user, would be collected at the beginning of the packet creation process, or if a marker would be used and a query to the database would be made each time a new packet was created. Figure 3 below illustrates the two designs that were being considered.

In design option one, all of the IP header entries are retrieved at the start and the double array holding the entries would be checked each time through to determine whether or not there were more entries, i.e. more packets, to send. Design option two in Figure 3, shows each IP header being retrieved one at a time. The Replayer then checks if an entry is returned each time through to determine when it has reached the last packets that match the options given by the user. The decision on whether or not to collect all of the IP headers at

once or to collect them one at a time, was put off until the packet creation module was created and in place.

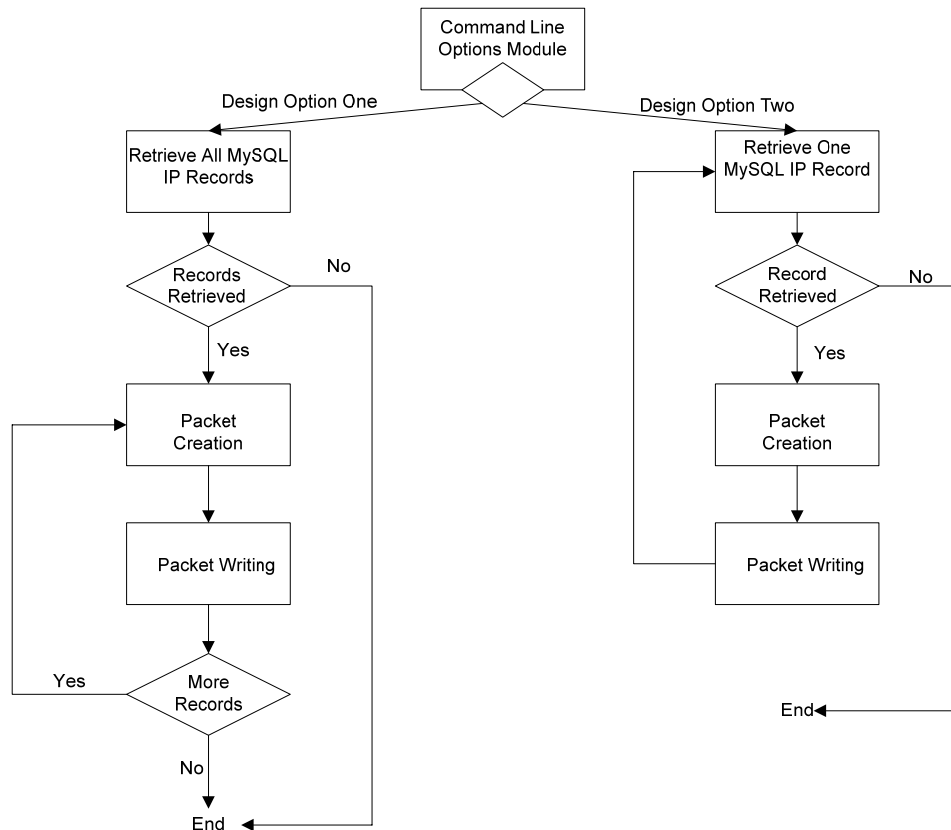


Figure 3. Two IP Header Retrieval Design Options

4.2.3.2 Packet Creation

In order to understand the packet creation process, it helps to start by looking at Libnet and also understanding where the packet creation functions were placed within the entirety of the Replayer as shown in Figure 2. Libnet itself is based on the idea of creating a packet from the ground up. For example, a TCP/IP packet would be created in the following steps; TCP options, TCP payload, TCP header, IP header, write packet to the wire. An amazing quality of the Libnet libraries is the amount of functions that are available to the user and the ease at which they can be used. They require only the information contained in

the headers and Libnet will build the packet from that information and then write it out to the network.

Thus, the first step in creating the packet was to grab the necessary information for each packet from the MySQL database. Using the queries developed in the MySQL module, the IP header information was retrieved first, and once the current header was successfully stored, it would be checked to see what protocol was being used. In order to keep things as simple as possible, only three protocols were supported from the start. These protocols were TCP, UDP, and ICMP. When the IP header was checked for the protocol, if it did not contain one of these three, then the Replayer would stop and generate an error letting the user know that an unknown protocol was detected. This never occurred however due to all of the attacks that were stored in the MySQL database falling under one of the three protocols mentioned.

After the IP header was checked for the protocol that was used, the Replayer would then jump to the correct packet creating function based on that protocol. That is, if a TCP protocol was discovered, then the program would go to the TCP creation functions. This was the same for UDP and ICMP. Once the Replayer had gone into a one of those three functions the next step was to then create the protocol headers and options that would be used by that particular packet.

In the case of a TCP, UDP, or ICMP packet, the method to create the packet headers was primarily the same. The IP header that was used for this packet would be matched up with its corresponding transport layer header, or network layer in the case of ICMP, from the MySQL database. This information would then be retrieved from the database using SQL queries and would be used to create the packet's transport layer header. The queries used in

the packet creation functions were developed much faster than the initial queries used to retrieve the IP header information at the beginning of the program. This allowed more work to be done on the actual implementation of the headers rather than spending several days modifying SQL queries to get the correct result back. Below is an example of the TCP function protocol that is used by Libnet to create the TCP header.

```
libnet_build_tcp (    u_int16_t    sp,          // Source Port
                    u_int16_t    dp,          // Destination Port
                    u_int32_t    seq,        // Sequence Number
                    u_int32_t    ack,        // Acknowledgement Number
                    u_int8_t     control,     // Control Flags
                    u_int16_t    win,        // Window Size
                    u_int16_t    sum,        // Checksum
                    u_int16_t    urg,        // Urgent Pointer
                    u_int16_t    len,        // TCP Packet Size
                    u_int8_t*    payload,     // Payload
                    u_int32_t    payload_s,  // Payload Size
                    libnet_t*    l,          // Libnet Handle
                    libnet_ptag_t ptag       // Libnet ID
                    )
```

Figure 4. Libnet TCP function prototype

As can be seen by Figure 3, the function requires all of the information that is present in a TCP header. Once all of those variables are passed into the function, it will correctly create a TCP header based on that information. The UDP and ICMP Libnet functions are both laid out in similar fashions. Before the packet's header could be created though, the payload for that packet would have to be retrieved from the database as well. The payload was not retrieved at the same time as the rest of the header information since they are stored in different tables. Therefore, before the header creation function was called, the payload was retrieved through a sequence of queries that would check if a payload existed and if it did, would return that payload to the packet creation function. A minor obstacle that occurred at

this point in the program was due to the formatting of the payload when it was returned from the database.

When using the SQL queries to retrieve the payload, the data was returned in ASCII format. This was because the payload was stored as ASCII in the database to represent the hex values of the payload. This made it easier for someone to peruse the database and the payloads, but did not work with the Libnet functions. In order to get around this, a small function was written that returned the equivalent hex value of each ASCII hex character represented in the payload. Once that hex value was returned, it was added onto the current payload through a series of bitwise operations. The character array that would hold the payload would start out as a NULL and with each pass through the conversion and transfer functions it would gain a byte from the MySQL payload. The end result of all this was a completed payload that was transferred from ASCII to hex, stored in a character array, and ready to be used in the Libnet functions.

Once the transport layer and ICMP protocols were finished, the next step in the creation process was the IP header. The IP header was created through a Libnet function by taking the information retrieved from the MySQL database along with the completed transport layer (TCP, UDP) or network layer (ICMP) header. The TCP, UDP, or ICMP header's size had to be determined as well as it was required by the Libnet IP function. Because the IP protocol encapsulates them, the headers are used as the payload for the IP function. Below in Figure 5 is the Libnet function protocol for the IP header. Once again it can be seen that it, similar to the TCP Libnet function, only requires the standard information of the header to correctly build it. When that was completed, the only step remaining was to write the packet out onto the network.

```

libnet_build_ipv4 (    u_int16_t    len,        // Length of Whole Packet
                    u_int8_t    tos,        // Type of Service
                    u_int16_t   id,        // Identifier
                    u_int16_t   frag,     // Fragment flags and offsets
                    u_int8_t    ttl,      // Time to Live
                    u_int8_t    prot,     // Protocol Number
                    u_int16_t   sum,      // Checksum
                    u_int32_t   src,      // Source IP address
                    u_int32_t   dst,      // Destination IP address
                    u_int8_t *  payload,  // Payload
                    u_int32_t   payload_s, // Payload Size
                    libnet_t *  l,        // Libnet Handle
                    libnet_ptag_t ptag     // Libnet ID
)

```

Figure 5. Libnet IP build function prototype

4.2.3.3 Packet Writing

Writing the packets out was the easiest part of the entire creation process but was still deemed to be a separate module in the Replayer's design once work on it began. This can be seen as the difference between Figure 1 and Figure 2 where the packet creation and packet writing module was split into two distinct code groups. Even with the ease that writing packets was accomplished with, it was not without its own set of difficulties. An important aspect of any tool that replays network traffic is timing. The timing of the replayed packets must match the timing of the original packets. This was made possible for the Replayer to accomplish due to Snort storing a timestamp into the MySQL database each time a packet was recorded. This gives the Replayer access to the times of each packet and more importantly, the time between the packets it will replay. In order to send the packets out with the correct timing, a simple set of timing controls were written. These controls had the Replayer check the current time and the time of the previous packet if one had been sent. The Replayer would then check the timestamp on the previous packet that had been sent and the timestamp on the current packet waiting to be written. From that point, the Replayer

would wait the appropriate amount of time that was left after subtracting the timestamps and the amount of time that had already passed since the previous packet had been written.

As for the actual function of writing the packets, while creating the packets, a Libnet tag is associated with each individual packet. After the IP header has been created, this tag is passed into another Libnet function that writes that packet out onto the wire. The Libnet tag is then released, and the Replayer jumps back up to check the protocol of the next IP header to be written, and starts the process all over again. The writing of the packets is the simplest part of the entire Replayer process.

4.2.3.4 IP Header Retrieval

With the packets at this point being created and written to the wire, it brought the groups attention back to the problem regarding the initial collection of the IP headers mentioned earlier. The group faced the decision as to whether retrieving all of the matching IP headers at once from the database was the best way to go, or whether it was more efficient to grab them one at a time from the database after each packet was created and written to the wire. With all of the IP headers grabbed at once, shown as design option one in Figure 3, it would relieve the Replayer from having to access the database for the IP headers each time a new packet was created and allow it to grab the next IP header from the double array where they w. Using design option one though would create significant wait time at the start of the program depending on how many IP headers matched the parameters given by the user and by how large the database in use was.

After several days of testing different types and amounts of packets, it was determined that grabbing each IP header individually instead of all at once, as shown in design option two in Figure 3, would work for the purposes of the Replayer. The only

concern with this method was whether the SQL queries would be too slow at returning the IP headers and disrupt the timing of the packets replaying. It was seen however, that the SQL queries were adequate and returned each entry in a timely manner.

With that completed, the rest of the semester for the senior design team was spent tracking down bugs and monitoring whether or not packets were actually being sent out. Testing was done briefly using ethereal [12] to monitor the network. From what was observed at the time, the Replayer was sending out packets, but was not sending out the correct payloads and would sometimes have incorrect headers. This was happening across TCP, UDP, and ICMP packets. Due to time constraints, no further work was done on the Replayer as the semester came to an end. The project was then passed on to me as a thesis project and I resumed work on the Replayer in June of 2005. Thus, phase one of development was complete and stage two of development began.

4.3 Stage Two

In June of 2005 I took over sole responsibility of the ICE project. At this point in time, a framework had been laid and the rudimentary beginnings of the Replayer were in place. This second phase of development would be the longest of the project and would last until August of 2006. During this time, numerous bug fixes would be implemented along with greater functionality and a more precise scope of what exactly the Replayer's goal was to be. Once again this section will cover Snort, MySQL, and the Replayer and the changes they went through during this phase.

4.3.1 Snort

The setup of Snort was not in question at anytime during the first stage of development. It correctly monitored traffic and stored suspicious looking packets into the MySQL database. Therefore, it was not looked at specifically when the next stage of development began. In fact, the first sign that there might be a problem with Snort did not arise until September of 2005. During the summer of 2005, I only devoted a small amount of time to the ICE project. When I was working on ICE, my time was spent tracking down bugs and researching SQL so I could better understand how to write a more optimized query to help speed up the Replayer. In August however, testing began on the Libnet functions to determine why some packets were incorrect and others were not. The specifics of those tests will be discussed in the Replayer section below. The results however, deal primarily with Snort.

4.3.1.1 Barnyard

It was discovered in September of 2005, that the replayed TCP packets contained payloads that were much larger than the MTU of 1500 bytes that is standard on the Ethernet protocol. Looking at the MySQL database it was also discovered that there were payloads of TCP packets stored within the database that were larger than the MTU. After some simple deduction, I believed that Snort was storing payloads that were much larger than was possible for a single packet to hold. This fact led to the discovery of the stream4 preprocessor, which was mentioned in Chapter 2 of this paper.

It can be recalled from Chapter 2 that the stream4 preprocessor is responsible for recreating the entire stream of the TCP connection. This allows the Snort software mechanism to check the entire payload of the stream for an attack. This keeps attackers from

splitting up attacks through different payloads. After looking through Snort's manual, it was an obvious fact that the stream4 preprocessor was a required preprocessor for Snort to run effectively and would therefore have to remain running. The obstacle then, was to determine how to break up the stored payload to allow for the correct replay of the packets to occur.

Before I did much work on fragmentation issue, it turned out, after a little research, that Snort had already solved this problem. Before going into the solution to this problem however, it is necessary to understand Snort's priorities when it is running. While Snort is running as a NIDS, it has two main functions; check all traffic it sees against the rule-set it has, and store all of the offending packets into the MySQL database. When Snort is given a high amount of traffic, or a large stream that it is reassembling, it can lead to packets dropping while it is writing to the database. Dropping packets means that it lets some packets go by without checking them for attacks. That is the main reason why, Snort would store only the entirely reassembled payload from a TCP stream. Reassembling the packet, checking it for an attack, and then taking the individual packets from the TCP stream and storing them into the database would take too much of Snort's time and processing power. This would then result in dropped packets. So to get around that problem, Snort only stores the entire reassembled payload.

The creator of Snort, Martin Roesch, therefore wrote an add-on called Barnyard that would handle writing to the MySQL database for Snort. This allows Snort to write its output in a language called fittingly the Snort Unified Language. It is a strictly binary representation of the data and allows Snort to spend a lower amount of its processing power on writing the output and more of it devoted to monitoring packets. This alone though, was not enough to solve the payload issue. It turns out that with Barnyard operational a single

option had to be enabled in Snort's configuration file. This option calls for the flushing of streams when an attack is detected. For example, when a TCP stream is being reconstructed by Snort, if an attack is shown to be held within that stream, the entire stream, meaning all of its individual packets, is flushed to the database. In the case with Barnyard, they're flushed into the Unified language and Barnyard then writes the individual packets into the MySQL database. With the configuration file modified and the Barnyard add-on working successfully, the problem with Snort logging larger than normal packets was solved by mid October of 2005. This turned out to be the only significant problem that presented itself, and the only changes made to the Snort setup throughout the rest of the second stage was updating its rule-set and operating version.

4.3.2 MySQL

During the second stage of development, the MySQL server underwent very few changes. It was upgraded as necessary to newer versions and patched when needed. The only major change occurred in June of 2006 near the end of this stage. During that time, queries were being used to search through the database to look for all of the different types of attacks and the IP addresses associated with those attacks. These queries would later on produce an output to the user that would allow them an easier way of seeing what they could replay. This process will be covered in more detail in the next section 4.3.3. With these queries though, came a minor change to the database setup. An index was added to the EVENTS table of the MySQL database. This allowed for the quick lookup of each attack that occurred rather than slowly going through each entry. By doing this, the queries were sped up from taking close to 5 hours to complete, down to 30 seconds. This was the only major change that the MySQL setup underwent for this stage of development.

4.3.3 Replayer

Starting in August of 2005, the first work to be done on the Replayer consisted of tracking down bugs that were left over from the first stage of development. Specifically, the main goal was to find out why the packets did not look correct when they were being monitored by ethereal. The first bug to be tracked down and fixed was a very minor one and was discovered in the payload building function of a UDP packet. During one of the bitwise operations that were used to build the payload, there was a double OR operation. This of course is the correct symbol if you are comparing two items within C but not for a bitwise operation. The double was changed to a single OR operation and the payloads were built correctly from then on. This solved the mystery as to why UDP packets were not sending correctly and focus then moved to the TCP packets.

4.3.3.1 Libnet and SQL Debugging

Earlier in section 4.3.1.1, the process of how the TCP packet payloads were discovered to be too large for a single packet was covered, and how, by varying Snort's setup, this problem was solved. Another small bug was also discovered in the payload building function for TCP packets. It was very minor and was discovered by simple line by line debugging. It turned out there was an extra two left in a line that essentially, doubled the size of the payload. Removing this, and modifying Snort, finally produced a correctly sized TCP payload. Both the UDP and TCP packet problems were solved by October of 2005. The ICMP packets never had any problems during the creation process.

Another early bug that was ironed out in September of 2005 was the SQL queries used to access the database. These queries were inefficient to begin with but were also discovered at this time to not return the correct entries in some specific situations. The

problem turned out to be a query based upon a specific port number would return the correct entries early on, but would skip those in the middle of the database and return entries from the end of the database instead. It was also discovered that this only occurred when the attacks it was retrieving were switching from UDP to TCP or visa versa. The bug was found to be an incorrectly written query. This query would grab the first entry it found that matched the port number given, be it UDP or TCP, and would continue to grab the correct entries until it came upon the other protocol as the next entry. From that point, it would grab the entry that occurred after all of the other previous protocol packets were passed. For example, a UDP entry is grabbed as entry number 94. A TCP packet is the next entry that matches that port and that entry is number 200. Also, consider that the last UDP entry matching the port number is 500. The Replayer would grab the correct entries up until entry 94. It would then see the TCP entry and jump to any entries after the last UDP packet at number 500. After several days of analyzing other queries and studying the layout and the goal of the query I wanted to create, it became obvious that the query currently in place was attempting to do a complicated query only using simplistic methods of the SQL language. The solution turned out to be a simple fix and the query code was rewritten. Appendix B shows the finalized query code that the Replayer uses currently.

The minor tweaks to the Libnet functions and the SQL queries lasted into October of 2005. Another change that occurred in October dealt with how the IP headers were grabbed from the MySQL database. In section 4.2.3 Figure 3 shows the two different design options that were in consideration for use. At the end of the first stage of development, option two was taken as there was no noticeable slow down when the individual IP header records were grabbed from the database in each pass through the loop. However, as the database grew

larger I found that there were delays in grabbing IP headers further back in the database. I spent several days trying to determine if there was a problem with the queries that was causing this delay. In the end however, I decided to change the implementation to design option one shown in Figure 3. This caused a small amount of wait time at the start of the program as all of the IP headers were collected, but was a small price to pay due to the fact that any and all delays that might occur due to queries during the actual replaying would disrupt the timing of the attack.

From that time, the ICE project was put on a hiatus as they Cyber Defense Competition at Iowa State University began and all of my efforts were put towards that endeavor. The next move forward on the Replayer design came in January of 2006.

4.3.3.2 Presentation to User

During January the focus of the work on the Replayer was moved away from the functionality of the replaying itself, to the presentation given to the user. At this time, the decision was made to stick with the command line interface that was in place. I also decided to expand the number of search options that the user would have and also give the user an option to receive a list of the attacks present in the database and the IP addresses that were associated with those attacks. This would allow the user to run the Replayer program to receive the necessary attack information to help them make a choice on what they wanted to replay.

The first step in this process was to decide what the exact type of information that was to be retrieved would be. I decided that the user would get a list of every individual source IP address along with a listing of the different types of attacks that came from that address. I also made the design decision of creating a double linked list that would store this

information from the database. The first linked list would contain all of the unique source IP addresses. This first list would be receiving its information from a simple SQL query that would retrieve each unique source IP address. Another SQL query would then return all of the attack signatures each IP address in the first linked list. The second linked list would then contain all of the attack signature numbers returned by the SQL query for each IP address. This second linked list would spawn off of each IP node. This seemed to be a simple endeavor but turned out to take several weeks to correctly implement, debug, and optimize.

Regrettably the initial code for the double linked list contained a bad memory address half way through the list and would crash the program each time it was run. It was a very minor bug and left no information on what exactly had caused the program to crash. Using DDD [13], a front end for GDB [14], I was able to track down the exact memory location and node in the linked list that was causing the program to crash. As it turned out, I had forgotten to set a pointer in the second linked list's attack signature nodes to NULL. When the program finally cycled through the linked list to that segment of code, the pointer was pointing to a random memory space outside the memory allocated for the program, which in turn caused a segfault and crashed the program. This took a little over a week to track and down due in part to my lack of familiarity with DDD and GDB.

After the segfault had been tracked down and corrected, the next issue at hand was the speed at which the linked lists were getting populated. The SQL query that were in place to retrieve the unique source IP addresses had no trouble executing in a sufficient amount of time. The SQL query that accessed the EVENT table to retrieve each attack signature however was extremely slow. It operated at such a slow rate that I first checked it to make sure that there was not a bug within the query itself. It was at this point in time, that it was

discovered that the MySQL database needed to be indexed for a faster search time. This was mentioned previously in section 4.3.2.

When the index was added to the EVENT table, the searches drastically decreased in time. The initial searches before indexing took over 5 hours to complete. When the EVENT table was indexed it took less than 30 seconds to populate the second linked list. Once this was accomplished, I turned my attention to the type of storage that would be used for this information.

With the database being populated from a single Snort session, the assumption was made that the database would not be changing once the session was complete and the user was ready to replay attacks from that database. Therefore, the user would only need to run the Replayer once with the option to retrieve the attack information. Thus, this information needed to be stored in a way that the user could access it quickly and easily without having to retrieve it from the MySQL database.

Several different options were considered when looking at different storage types. In the end it was narrowed down to three choices; a separate MySQL database, a comma separate value (CSV) file, or a text file. The MySQL database was quickly thrown out as a plausible idea because it would still need to be accessed for the stored information. The exact reason of storing the information in the first place was to avoid connecting to the database so the separate MySQL database was discarded. The CSV and text file options ended up being used in conjunction. The text file was going to be used by storing the IP addresses and each attack associated with that IP on a single line. Then each sequential line would hold the next IP address. I also decided that separating the IP addresses and attacks by

a comma would make it easier for me to access the information through the Replayer. This in turn, made the text file into a CSV file and I went from there.

Code was put in place to enable the Replayer to notify the user that a file already exists with attack information when the user runs the option to retrieve attack information from the MySQL database. The user then has a choice of using the CSV file that contains the information previously gathered, or overwriting that file with new information from the MySQL database. Creating these functions and correctly implementing them took several months in the spring of 2006. During that time other projects began and my time to devote to the ICE project dropped dramatically. Work stopped on the Replayer from April of 2006 and started again in June of 2006. This was also effectively the end of the second phase of development.

4.4 Stage Three

The beginning of the third stage of development began in June of 2006. At this time, Snort and the MySQL database were both functioning correctly and the main focus of development was on the Replayer. At the end of stage two, the collecting and organizing of the IP addresses and the attacks had been completed and the user now had the ability to look through that information to allow them to select the appropriate attacks to replay. From this point, my focus would be on developing more options for the user, reorganizing the Replayer source code to a more object oriented design, and in depth testing.

4.4.1 Replayer

The options available to the user, was a large concern at the start of the third and final stage of development. At that time, the only options that had been written for the user were

selecting the source or destination IP address of an attack or the port that an attack was run on, or retrieving a list of IP addresses and the attacks associated with those IP addresses. From this point several more options were added that were required to bring the Replayer up to specification with its design goals. The main options added gave the user the ability to rewrite the source and destination IP addresses of the packets to be replayed. Another two options that were to be created would give the user the ability to quickly see a list of captured attacks and to replay packets based on attacks.

The ability to rewrite the source and destination IP addresses was one that was set to be implemented in the Replayer from the very beginning of the project. The option to do so was not implemented until this stage of development due to other factors of the ICE project being of a high priority. When I did start to code these options into the Replayer, it turned out to be a very simple job to implement and I was able to complete it and debug it in a matter of days. The next option I added showed them the attacks that were currently stored in the database. This was completed very quickly and only required to query the database and check the attack signatures that were stored in it. Once that was accomplished, the final option I wrote in allowed the user to choose to only replay packets that were involved in a specific attack. For example, if an attack had a signature number of 15 in the MySQL database, then choosing to replay packets that matched that signature would replay all of the packets in the database that matched, regardless of the source or destination IP addresses involved. This was done to give the user the ability to send out a blanket of packets; all of the same attack type, rather than just a few that were involved with a specific IP address.

Once these options were implemented, they were tested and several minor bugs, which dealt with the format of the IP addresses, were tracked down as they were discovered

over the next few weeks in July of 2006. When that was completed, the Replayer at that point in time had the ability to retrieve specific attacks based on IP address, port, or attack signature and rewrite the source and destination IP addresses of those packets. It was at this time that a major oversight was discovered in the way I had created the options. In stage two of development, I allowed the user to create a file that contained all of the unique IP addresses and the attacks associated with each respective IP address. Up until this point in time though, I had not given the user a way to use that information to it fullest extent and this became obvious to me as I looked over the options I had implemented thus far.

When the user would run the Replayer and select a source IP address, the Replayer would grab all of the packets that match that source IP address from the MySQL database and then replay those packets out onto the wire. If the user selected a specific attack as an option then the Replayer would grab every packet that matched that signature and replay all of the packets out onto the wire. I had not however, written the option code to allow the user to use both of these options at once. This was a huge problem that I had created for myself and became apparent immediately when I tried to run the Replayer with the options for a source IP address and an attack number. The Replayer grabbed all of the packets matching the source IP address and proceeded to ignore the attack number.

When I realized what had happened, I began to read over the options code in the Replayer. In this code, I realized that I had only allowed for one option to be selected, and when the code reached the first option that matched the one given at the command line, it would exit from that function and continue to the next section of the Replayer and start to create the queries for the MySQL database. The fix for this problem at first appeared to be very straight forward to me. The function `getopt ()`, which handles reading in the options,

would run through the state and case clauses multiple times, rather than just once, until it had reached the end of the options present in the command line.

The unfortunate side effect of this change became quite evident when the queries to be executed were looked at. All of the queries had been designed to only run once. With the change to the `getopt()` option handling in effect, it made the current implementation of the queries obsolete and they had to be worked over. After looking over the code, I implemented several variables that would act as counters. These counters would keep track of the number of options that were passed in from the command line. I then added another segment of a query whose sole purpose was to combine two or more of the previous query statements that were coded in. This enabled the Replayer to grab the options from the command line, count how many options there were, take each of those options and create the correct query for each of them, and then combine those separate query statements into one single statement and access the database. The original queries did have to have a few minor adjustments made to them in order for them to work as part of a larger one if there was more than one option, but nothing that was changed affected the performance of them.

4.4.1.1 Code Cleanup

Once the ability for the user to select multiple options had been correctly implemented, I decided to spend the next month going over the Replayer code in detail and reorganizing it. The experience with the SQL queries made me realize that their might be other gross and obvious oversights that I had created in the code. Another reason I had was a direct result of the way the Replayer had been written. There were many times that coding on the Replayer did not occur for several months and each time I came back and continued coding I would be working on a different section of the Replayer. This had lead to a working

Replayer but it was rather disjoint in the way the code was held together. Due to this, it was becoming exceedingly obvious that implementing new functionality in the future would be an extreme hassle. It was for these reasons that reorganizing the code, geared specifically towards a more object oriented design, seemed to be in the best interests of the Replayer and its future developers.

One of the first parts to be reorganized was the global variables. Many of these were used to create the IP header, the MySQL database connections, and the Libnet headers and had been created at the very beginning of the Replayer. At that moment, I had moved several sections of code off of the main source file and into their own source file. With this change, having the global variables in such a free standing state hindered the reorganization of the source code due to conflicts in access to the variables. To combat this, I decided that these initial variables would serve their purpose much more efficiently and cleanly if they were grouped into their own separate structures and source file. Once that was accomplished, it was much easier to move about the code and pass a structure into the various Libnet functions rather than passing an enormous amount of separate variables. Appendix C shows the global variables before the code clean up and the newly created structures in use after the clean up.

4.4.1.2 MySQL Configuration File

Once the Replayer had undergone the code reorganization, it was a much cleaner looking program. It had been changed to be much more module and made adding additional playback functionality much easier than before. At this time, the idea of configuration files being used for the MySQL database connection started to surface. During the reorganization of the code, I noticed that the MySQL database information was hard-coded into the

Replayer. This meant that in order to have the Replayer connect to a different MySQL database, the lines of code containing that information would have to be modified and the Replayer itself would have to be recompiled.

I decided at that point in time, that a configuration file was desperately needed for this information as it was inconceivable to require the user to recompile the Replayer each time the database might change. After reviewing exactly what information the Replayer needed to connect to the database, I implemented code that would read in a configuration file of my design. This configuration file would hold the server's hostname, the username, the password, and the database to connect to. Example entries in the configuration file can be seen in Appendix D. During the development of this simple configuration file, the idea that more than one server might be in use by a user was taken into consideration. Therefore, the configuration file can contain more than one server entry and each entry is recognized by the number at the beginning of an entry. In this way, the user can specify at the command line what entry to use and can store multiple databases' connection information in one configuration file.

A small problem that developed from this additional option was how it was executed. Before this option was in place, the first action that the Replayer would execute would be the code to connect to the database. After much mulling, I determined the best course was to split up the area that handles the options. It was my intention that by doing so would allow the Replayer to grab the configuration file option, connect to the database, and then resume its normal process of checking for additional options that were used on the command line. Figure 6 below shows the different processes the Replayer used before and after the change.

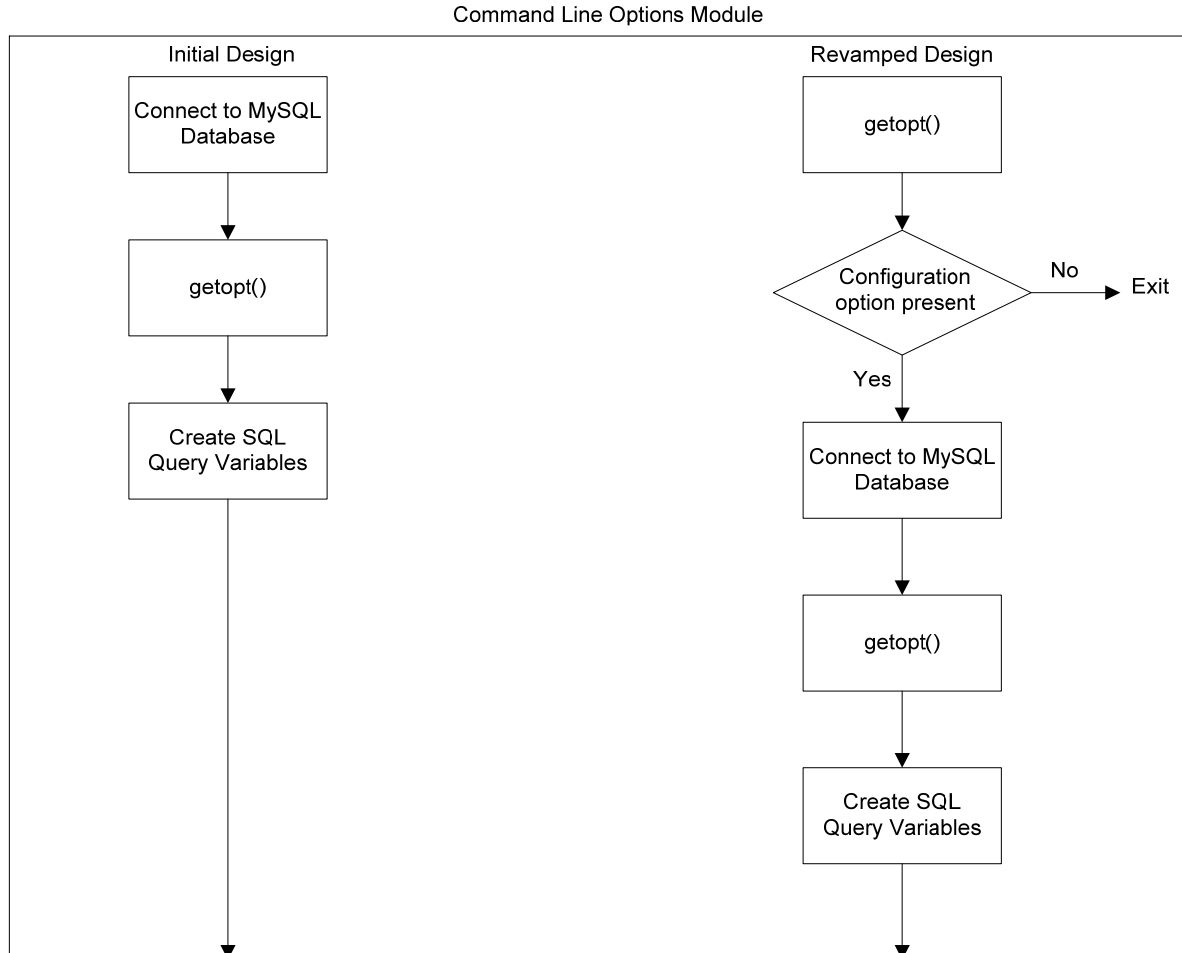


Figure 6. Initial and Revamped Option Retrieval and Database Connection Designs

4.4.1.3 TCP Three-Way Handshake

At this point of development on the Replayer it was December 2006. After a small amount of testing, which will be covered in detail in the next chapter, I discovered a fatal flaw in attacks that were carried out over a TCP stream. More specifically, attacks that involved two way communications between the Replayer and the target would fail 100 percent of the time. These types of attacks would involve the target machine running any service that requires two way communications, and the attack itself would have multiple packets to it. The failure was caused by the Replayer's lack of ability to correctly respond to

the target. This also brought up an error that was not recognized in the way Snort handles TCP stream attacks until this testing occurred.

When Snort flushes and stores the TCP stream from an attack that it has monitored, it does not include the three-way handshake from the stream. A three-way handshake is the method that occurs during the initialization of a TCP connection. This led me to realize that when the Replayer would replay a TCP stream attack, it would fail immediately due to the Replayer and the target not creating a successful handshake for the exchange of the attack packets. I also discovered that the Replayer had an inherent flaw when replaying TCP stream attacks even if the handshake was stored with the stream. Given that the Replayer had made a successful connection with a target machine through the three-way handshake, the acknowledgment numbers of the Replayer's packets would be the same as they were from the MySQL database. In order for a TCP packet to be accepted by the target however, the acknowledgment numbers must match the sequence numbers that were given by the target for that particular connection. I had not taken this into account when designing the replaying functionality of the Replayer and therefore every packet that the Replayer would send over the wire to a target via a TCP stream would be rejected and the target would try to reset the connection.

After two days of going over the Replayer's code and considering my design choices, I decided to implement another option that would allow for the Replayer to connect via a three-way handshake and modify the acknowledgement numbers of its packets to correctly match the sequence numbers being sent from the target. My decision to make this an option for the user was made by considering that not all of the attacks that the Replayer will send will be TCP based, and that not all of those TCP attacks would involve multiple packets.

Therefore, it would be up to the user to decide when to have the Replayer initiate a three-way handshake and modify the acknowledgement numbers for a TCP attack.

The design and implementation of this option was by far the most involved of the project. It meant that not only would the Replayer be replaying packets out to a target, but it would also be monitoring traffic from the target for specific information, such as the acknowledgement numbers. This brought timing concerns to the forefront of my mind when I began to develop the code and would be the cause of most of the problems in correctly creating the functionality.

The first step in the implementation process of this option was to design the three-way handshake. This would be handled in a new function called IceTWH. For IceTWH to work, the Replayer would send out an initial TCP packet correctly formatted with the SYN flag set to act as the first packet in a three-way handshake. The target would receive that packet and reply to the Replayer with the second packet in the handshake that has the SYN and ACK flags set. The Replayer would then grab that packet from the network, locate the sequence number of the packet, and reply back to the target with that sequence number as the third packet's acknowledgement number. This third packet would also have the ACK flag set. The design of the three-way handshake was fairly simple. I found out that the implementation of it would not be.

To accomplish the task of the three-way handshake, the libpcap library, which was mentioned briefly in Chapter 2, would be used to build a limited traffic sniffer. The objective of this traffic sniffer would be to grab the second packet containing the SYN and ACK flags from the target and to extract the sequence number from it. Before that was done though, the initial SYN packet was created using Libnet. I made this packet static, in that there is no

dynamic information contained within the packet except for the target's IP address. All of the TCP options in this packet were hard coded into it and are not meant to change. The standard TCP build function from Libnet was then used along with the TCP build options function to create the packet. This permitted me to correctly build a standard first stage SYN packet for the three-way handshake.

Once that was accomplished, I used the tutorial [15] from the creator of Tcpcap as a primer to writing my own sniffer. The initial difficulty I had with creating the sniffer had less to do with the syntax of libpcap and more to do with the timing and placement of it in relation to the Libnet functions that were responsible for creating and sending out the three-way handshake packets. My original version of the IceTWH function had the initial SYN packet created and sent out to the target. After that first packet was sent, the sniffer then initialized itself and started to monitor the network. I found out quite quickly though that the initialization time of the libpcap library and its corresponding functions, requires a longer amount of time than it takes the target machine to reply back to the Replayer with the second packet of the three-way handshake. This resulted in the reply packet being missed by the Replayer, the IceTWH function would exit due to the failure of the sniffer, and the replaying of the attack would continue on from that point and fail.

When the timing issue presented itself, I began to consider whether to fork off or thread the process of sniffing from the main three-way handshake function in order for it to run simultaneously and monitor the network correctly. As it unfolded though, the real solution was very simple and easy to implement. It was the initialization of the libpcap functions and variables that created the timing issue mentioned previously. By moving the initialization segment of code to the start of the IceTWH function, I was able to create the

SYN packet, and then call the sniffer code immediately afterwards without any delay. With the initialization process already taken care of, the sniffer actually picked up my initial SYN packet as well it was so fast. This solved the timing issue I had and allowed the Replayer to grab the correct sequence number from the target's SYN and ACK packet reply.

From that point, I took the sequence number and put it into the third and final packet of the handshake as the acknowledgement number. That packet was then created and sent to the target to complete the handshake. It was then a matter of monitoring the packets that were returning and modifying the acknowledgement number on the Replayer's packets by the correct amount for the size of each payload. I should also note that during the testing of this section of the Replayer's functionality I was spoofing a live machine on the network. This resulted in a race condition between the Replayer and the live machine that was being spoofed and would cause a 33 percent failure rate in the Replayer due to the TCP connection being disrupted. However, during a test without spoofing a live machine on the test network, the success rate was between 90 and 100 percent. Work that needs to be completed on this section is discussed in Chapter 6 FUTURE WORK.

CHAPTER 5. TESTING

Testing of the various components of the ICE project was ongoing through out the development process. Tests of the Snort and MySQL configurations were necessary with their initial setup and when bugs were discovered. Testing of the Replayer was conducted ad-hoc and the many functions contained within the Replayer were debugged and optimized during development. Near the end of the third development stage a testing environment was setup to ensure that, along with the packets getting replayed correctly from the Replayer, the packets were also replicating the attack correctly against the target machine.

5.1 Testing Methodology

I already had confirmed from my testing done in the previous stages that the Replayer did in fact replay the attack packets and that those packets pushed out onto the network. The main question I had was whether or not, given an identical target machine and environment, that the Replayer could trigger an actual attack. In order to accomplish this, I would need a test environment that would allow me to play out an attack against a target, monitor and capture that test using Snort and MySQL and then replay the same attack against the same target using the Replayer. By following these steps, I would be able to monitor the target machine and observe whether or not the Replayer was capable of causing the same end result on the target machine as the original attack had done. Before I could start however, I needed to create my test environment.

5.2 Test Environment

The test environment that I planned to setup would contain three machines. One of the machines would contain the Replayer and the MySQL database for the Replayer. The second machine would contain Snort and the third machine would be a standard Windows XP box and act as a target. When considering how to setup these machines, it came to my attention that I could make use of the VMWare [16] application in this scenario.

I had previously worked with fellow graduate student Nate Karstens on a presentation for the Information Assurance Student Group [17] and had helped to create a disk image that contained VMWare for that presentation. On that particular image, there were four operating systems in use. The first operating system was Fedora Core 4 and was the host operating system that ran the VMWare software and tools. The other three operating systems were all virtualized and included a Windows XP box, a FreeBSD 6.1 system with Snort installed, and a FreeBSD 6.1 system. I worked and modified the image for a day and decided that it would suit my needs as a test environment. The figure below shows the layout of virtualized network. To avoid any confusion, I have labeled the Windows XP box as machine T since it is the target machine, the FreeBSD boxes as machines S and A for Snort and Attacker respectively, and the Fedora Core 4 as machine H because it is the host.

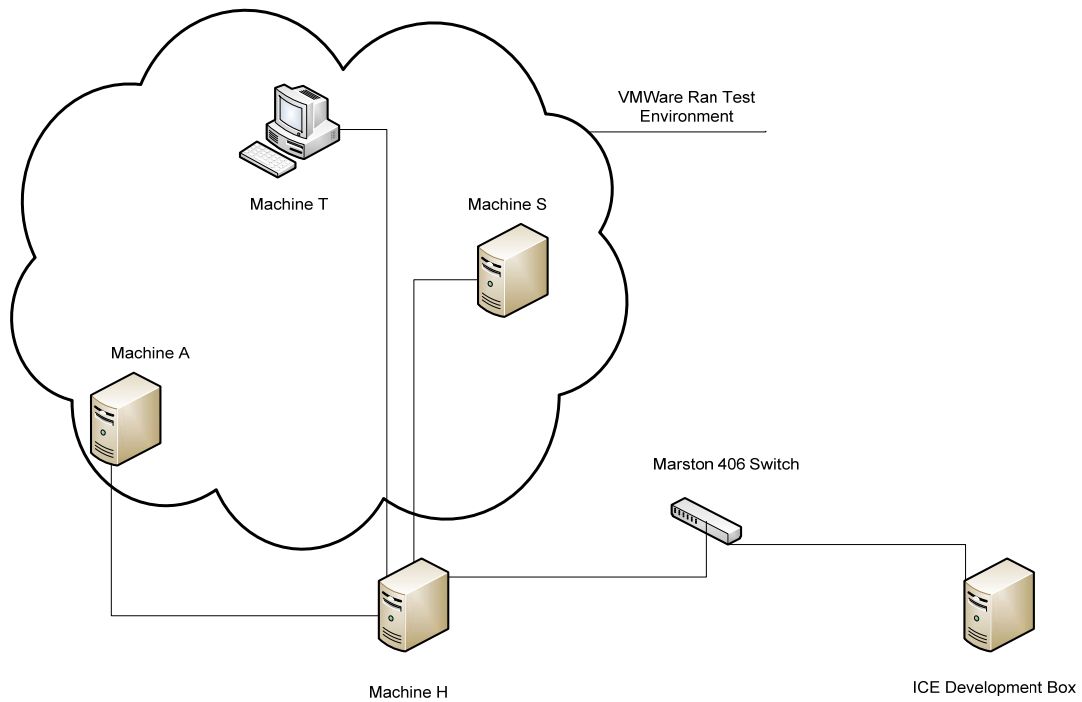


Figure 7. Test Environment

I then added machine H to the same network that the ICE development machine was currently sitting on. Once that was finished, it was a two step transfer to move a copy of the Replayer executable from the development machine over to machine A. I transferred the executable using SFTP from the development machine to machine H and then used SFTP again to transfer it from machine H to machine A. This allowed me to make modifications to the Replayer, recompile it, and then quickly transfer the new executable into the test environment.

Machine A also had Snort and a MySQL database installed onto it. These, along with the Replayer, turned it into a full version of ICE and allowed it to monitor the test environment's network for attacks and store them into the MySQL database. These attacks could then be replayed by the Replayer running on machine A. I also installed Snort onto machine S. This created an independent Snort machine that would be used to confirm that

the replayed attack packets would trigger the IDS. Machine H was also given Metasploit [18] which would allow me to send pre-made attacks to machine T.

5.3 Test Execution

With the test environment setup and full functional, I began to go through the Metasploit database and look for an attack that would be advanced enough to test the full functionality of the Replayer, but not so advanced that it was beyond what the Replayer was designed for. Most of the attacks stored in the Metasploit database are fairly simple, one shot attacks and would do nicely for a test. I finally decided upon the RPC DCOM exploit [19] that was known to help transfer the blaster worm [20] in mid 2003. The exploit in Metasploit was a simple code execution that used the RPC DCOM exploit to gain access to the target Windows XP machine. With the exploit chosen, I went about setting up for the actual test.

The first step I took was to initialize Metasploit on machine H. Once Metasploit was up and running, I traversed through the web based GUI and selected the RPC DCOM exploit and entered the IP address of machine T. Before executing the exploit though, I turned on Snort and MySQL on machine A so I could capture the attack packets and store them for the Replayer. When I was confident that machine A was running correctly, I executed the RCP DCOM exploit against machine T. Within several seconds of the exploits execution, a shutdown warning box appeared on the screen and machine T rebooted in several seconds.

The exploit had worked correctly and machine T had to restart. The logs from machine A also indicated that Snort had successfully captured the attack and that those attack packets had been stored in the MySQL database. At this juncture of the testing, I turned on Snort on machine S. This would allow an independent monitoring of the Replayer's packets

and would, hopefully, show the exact same warnings and captured packets that occurred with the original attack.

Once machine T was back online, I ran the Replayer with the commands that told it to replay all attacks in the database with the destination address of machine T. This had the desired effect of causing the Replayer to replay the entire attack stream that had been stored in the MySQL database. Once the packets had been replayed, I checked machine T and was greeted with the standard screen and no shutdown warning. The replayed attack had failed to generate the same end result as the original attack. However, machine S which was running Snort, did capture all of the packets and the replayed attack had triggered all of the same warnings. It was at this point in time that I did some research into the attack, and became aware of the fact that Snort was not recording the initial three way handshake of the TCP attack stream being used, and that the acknowledgement numbers would have to be modified in order for the attack to be replicated successfully. I took these findings and went back to work on the Replayer to make the necessary modifications, all of which has been documented in Chapter 4 section 4.4.1.3. It should be noted here however, that the testing environment was used extensively during this time period to help with the development of the TCP handshake module.

When the development of the TCP hand-shake module had concluded, I transferred the latest copy of the Replayer over to machine H and then on to machine A. I then ran the Replayer with the same commands as I had with the initial test and the results were satisfactory. Machine A successfully connected to the target machine T, and executed the RPC DCOM exploit on it with out trouble. This caused machine T to reboot and machine S also recorded the attack successfully. Due to time constraints, this was the only exploit

tested extensively with the Replayer. It showed however, that the Replayer could not only simply replay the packets onto the network, but when given the same type of target machine, and in the cases of these tests the exact same target machine, that the attacks would play out correctly.

Chapter 6. FUTURE WORK

There are two main areas of ICE, both within the Replayer, that require attention by future developers; the graphical user interface (GUI) and the three-way handshake module. The GUI aspect of the Replayer has not left the design stage and no actual code has been written to accomplish this task aside from several test programs to understand GTK, the graphics library being used for it. The three-way handshake code correctly functions at this moment but has timing issues and is not written in a very efficient manner at this point in time.

6.1 Graphical User Interface

The GUI for the Replayer has been briefly looked at. It will consist of a simple front end that will allow the user to input the target IP address to which the replayed attacks will be sent, a list of the attacks available to select from and their corresponding original source IP addresses, and the ability to select attacks to send based on the ports used. This will give the user access to all of the options that they would have if they were running the Replayer from the command line. In fact, the GUI will simply be a translator in which it will take the user's input and create the corresponding command to be executed on the command line. This should allow for the Replayer code itself to remain untouched and only require an understanding of what options the Replayer takes as inputs in order to develop the GUI.

6.2 Monitoring Sequence Numbers

Part of the code within the three-way handshake function uses libpcap to create a sniffer to monitor the incoming reply packets from the target machine. The implementation of this section of the code is very inefficient due to the sniffer being called after each packet is written out by the Replayer. The sniffer then monitors for the packet that is sent in reply to the Replayer's last sent packet. The sequence number from the packet grabbed by the sniffer is then taken and used as the acknowledgment number in the next packet sent by the Replayer.

A cleaner way to do this would be to create a thread for the sniffer that would always be running during an attack that had invoked the three-way handshake option. This would allow for the Replayer to avoid calling the sniffer after each packet had been sent and remove the possibility that the sniffer would miss the target's reply packet. In section 4.4.1, it was covered that the sniffer invoked during the three-way handshake did not miss the reply packet and was indeed so fast that it picked up the initial SYN packet sent by the Replayer. Even with this information however, it would still serve the Replayer better to have the sniffer handled in its own thread instead of constantly being recalled to operate between each packet sent by the Replayer.

6.3 MySQL Database Indexing

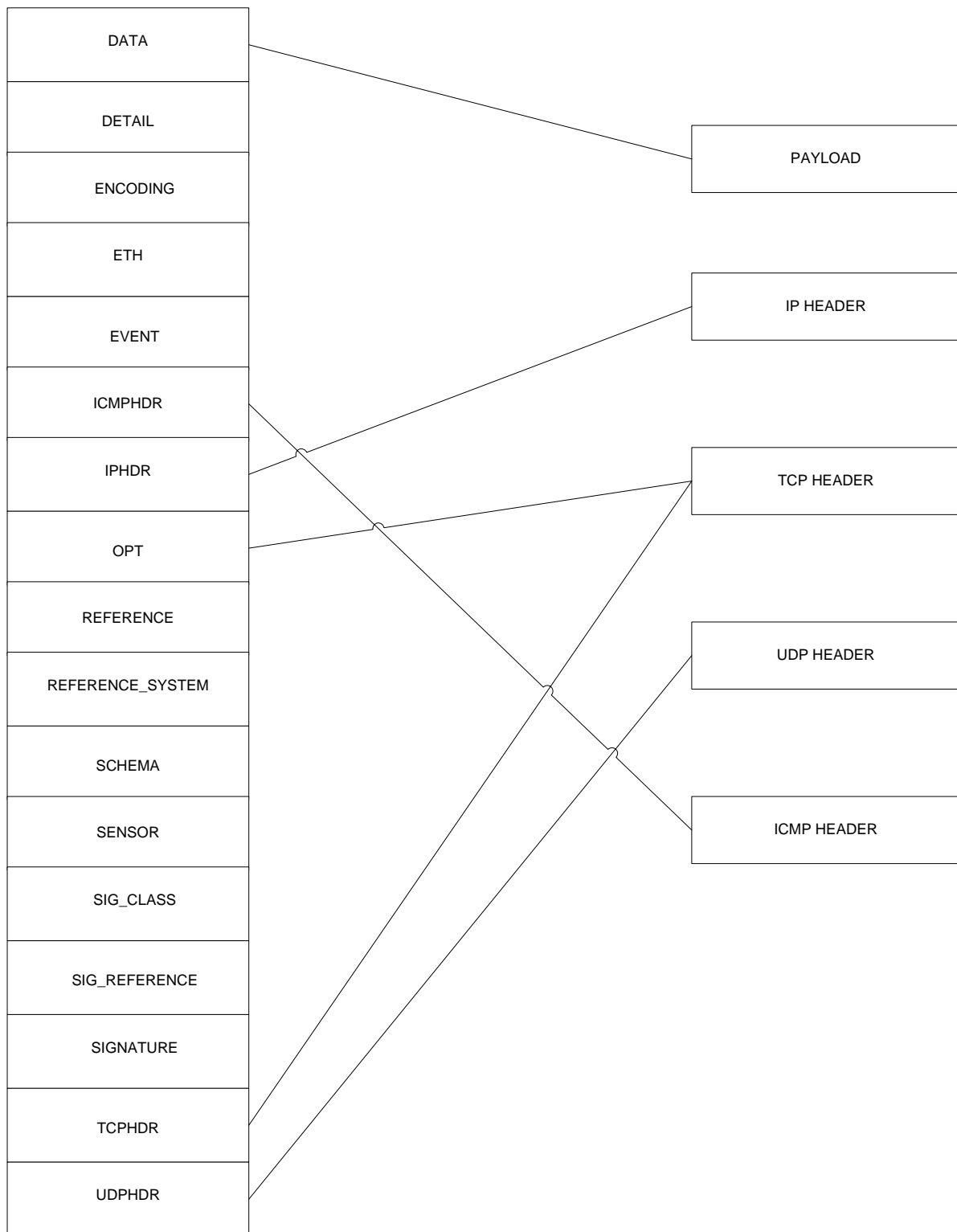
Another possible cause of the latency from the MySQL database could be the fact that the IP, TCP, and UDP header tables are not indexed. As was mentioned in section 4.3.2, the EVENTS table was indexed to provide faster searching capabilities when producing the attack information for the user. By indexing the IP, TCP, and UDP header tables, a similar

result should be expected to occur. This would allow for design option two from Figure 3 to be utilized and eliminate the initial startup time for the user.

CHAPTER 7 CONCLUSION

The progress made on the ICE project up to this point has created a solid foundation on which later developments and refinements of the software can occur. The main functionality is in place and each component has reached a point that I consider satisfactory. The Snort setup and initialization correctly monitors and stores every packet from an attack stream that it detects. The MySQL database is quick and efficient at handling data requests and holds all of the data from the Snort output. The Replayer at this time correctly takes the information stored in the MySQL database and replays the attack onto a network.

The Replayer has accomplished what it was designed to do and has achieved its goal. That is not to say that there are not improvements that can be done to it. As talked about in Chapter 6, there are a few issues that need to be worked on in the future. Also, the ability for the Replayer to handle more complicated attacks and perhaps handle data not retrieved from a Snort output could be looked at as well. This would allow the Replayer to be used with other attack data as long as that data was stored in a MySQL database. However, as stated in Chapter 1, the area of replaying attacks is large and holds many tools that can help replay attacks in various ways. The Replayer covers Snort-based output nicely and that may be all it ever is required to do.

APPENDIX A. MYSQL TABLES

APPENDIX B. SQL Code

```

while(x > 0){
    y++; //increase for each choice
    switch(selection[y]){
        case S_PORT:
            sprintf(spq1, "tcpdr.tcp_sport = %s", choice[y]);
            sprintf(spq2, "udphdr.udp_sport = %s", choice[y]);
            break;
        case D_PORT:
            sprintf(dpq1, "tcpdr.tcp_dport = %s", choice[y]);
            sprintf(dpq2, "udphdr.udp_dport = %s", choice[y]);
            break;
        case IP_SRC:
            sprintf(isq1, "iphdr.ip_src = INET_ATON('%s')", choice[y]);
            break;
        case IP_DST:
            sprintf(idq1, "iphdr.ip_dst = INET_ATON('%s')", choice[y]);
            break;
        case ATTACK:
            atk = atoi(choice[y]);
            break;
    }

    x--; //decrease to track the number of choices left to take
}

sprintf(query, "SELECT iphdr.* from iphdr WHERE ");

while (y > 0){
    switch(selection[y]){
        case S_PORT:
            sprintf(where, "( iphdr.cid IN ( SELECT tcpdr.cid FROM tcpdr
WHERE %s) OR iphdr.cid IN ( SELECT udphdr.cid FROM udphdr WHERE %s))", spq1,spq2);
            sprintf(query, "%s %s ", query, where);

            break;

        case D_PORT:
            sprintf(where, "( iphdr.cid IN ( SELECT tcpdr.cid FROM tcpdr
WHERE %s) OR iphdr.cid IN ( SELECT udphdr.cid FROM udphdr WHERE %s))", dpq1,dpq2);
            sprintf(query, "%s %s ", query, where);
            break;

        case IP_SRC:
            sprintf(where, "%s", isq1);
            sprintf(query, "%s %s", query, where);
            break;

        case IP_DST:
            sprintf(where, "%s", idq1);

```

```
        sprintf(query, "%s %s", query, where);
        break;

        case ATTACK:
            sprintf(where, "( iphdr.cid IN ( SELECT event.cid FROM event
WHERE event.signature = %d)", atk);
            sprintf(query, "%s %s", query, where);
            break;
    }
    if(y > 1) //this next line is to append several queries together for multiple options
        sprintf(query, "%s AND ", query);
    y--;
}
```

APPENDIX C. GLOBAL VARIABLES AND STRUCTURES

Before Code Cleanup

```

////////////////////////////////////
////////Global Variables////////////////////////////////////
////////////////////////////////////

//MYSQL//
MYSQL *conn, *TCP_conn, *UDP_conn, *ICMP_conn, *TIME_conn;
MYSQL_RES *ip, *attk, *time_res;
MYSQL_ROW ip_row, sig_row, time_row;
int cidNum = 0;

//LIBNET//
char errbuf[LIBNET_ERRBUF_SIZE];
libnet_t* l;
libnet_ptag_t t;

//std//
int size;
char ip_Sadd[17], ip_Dadd[17];
bool checkipS, checkipD;
int subPacket_s;

//IP INFO STORAGE//
u_long IP_src;
u_long IP_dst;
u_short IP_ver;
u_short IP_len;
u_char IP_tos;
u_short IP_id;
u_short IP_flags;
u_int IP_off;
u_char IP_ttl;
u_char IP_proto;

```

After Code Cleanup

```

typedef struct g_mysql{          //Struct holds the MYSQL connection information

//MYSQL//
    MYSQL *conn, *TCP_conn, *UDP_conn, *ICMP_conn, *TIME_conn;
    MYSQL_RES *ip, *attk, *time_res;
    MYSQL_ROW ip_row, sig_row, time_row;
    char *server, *user, *password, *database;
    char serv[20], use[20], pass[20], data[20];
    int select;
}mysqlglob;

```

```

typedef struct g_libnet{
    libnet_t* l;
    libnet_ptag_t t;
    int cidNum;
    int subPacket_s;
}libnetglob;

typedef struct g_address{           //Struct for the IP address
                                   //along with values to store user input address
    char ip_Sadd[17], ip_Dadd[17];
    bool checkipS, checkipD;
    u_char opt_t[1000];
    u_long AckHold;
    u_short PortHold;
    int option_s;

    ///IP INFO STORAGE///
    u_long IP_src;
    u_long IP_dst;
    u_short IP_ver;
    u_short IP_len;
    u_char IP_tos;
    u_short IP_id;
    u_short IP_flags;
    u_int IP_off;
    u_char IP_ttl;
    u_char IP_proto;
}addressglob;

typedef struct g_pcap{             //libpcap struct used for sniffing
    pcap_t *handle;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    bpf_u_int32 mask;
    bpf_u_int32 net;
    struct pcap_pkthdr header;
    const u_char *packet;
    int num_packets;
}pcapglob;

mysqlglob mytest;                //Declaring these structures here
addressglob addytest;            //allows them global access across the program
libnetglob libtest;              //removing problems with pointers and scope issues
pcapglob pcaptest;

```

APPENDIX D. EXAMPLE MYSQL CONFIGURATION FILE

1

server-localhost
user-snortman
password-ISEAGE
database-snort

2

server-localhost
user-snort
password-ICEMAN
database-snorttest

BIBLIOGRAPHY

- [1] “ISEAGE Internet-Scale Event and Attack Generation Environment” 12 March 2007.
<<http://www.iac.iastate.edu/iseage/>>.
- [2] “Cyber Defense Competition CDC” 30 March 2007.
<<http://survey.iac.iastate.edu/HSCDC/>>.
- [3] “Snort - the de facto standard for intrusion detection/prevention” 12 March 2007.
<<http://www.snort.org/>>.
- [4] “MySQL AB :: The world's most popular open source database” 12 March 2007.
<<http://www.mysql.com/>>.
- [5] “Tcpreplay – Trac” 12 March 2007. <<http://tcpreplay.synfin.net/trac/>>.
- [6] “SourceForge.net: The libpcap project” 12 March 2007.
<<http://sourceforge.net/projects/libpcap/>>.
- [7] “The Million Packet March” 12 March 2007. <<http://www.packetfactory.net/libnet/>>.
- [8] “Snort - the de facto standard for intrusion detection/prevention” 12 March 2007.
< http://www.snort.org/about_snort/>.
- [9] “GNU General Public License - GNU Project - Free Software Foundation (FSF)”
12 March 2007. <<http://www.gnu.org/copyleft/gpl.html>>.
- [10] Weidong Cui, Vern Paxson, Nicholas C. Weaver, Randy H. Katz, “Protocol-Independent Adaptive Replay of Application Dialog”, *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.
- [11] “Getopt – The GNU C Library” 12 March 2007.
<http://www.gnu.org/software/libc/manual/html_node/Getopt.html>.
- [12] “Ethereal: A Network Protocol Analyzer” 12 March 2007.
<<http://www.ethereal.com/>>.
- [13] “DDD – Data Display Debugger” 12 March 2007.
<<http://www.gnu.org/software/ddd/>>.
- [14] “GDB: The GNU Project Debugger” 12 March 2007. <<http://sourceware.org/gdb/>>.

- [15] Tim Karstens, “Programming with pcap” 20 March 2007. <<http://www.tcpdump.org/pcap.htm>>.
- [16] “VMware: Virtualization, Virtual Machine & Virtual Server Consolidation – VMware” 12 March 2007. <<http://www.vmware.com/>>.
- [17] “Information Assurance Student Group” 20 March 2007. <<http://iasg.ece.iastate.edu/>>.
- [18] “The Metasploit Project” 20 March 2007. <<http://www.metasploit.com/>>.
- [19] “Microsoft Security Bulletin MS03-026” 20 March 2007. <<http://www.microsoft.com/technet/security/bulletin/MS03-026.msp>>.
- [20] “W32.Blaster.Worm” 20 March 2007. <http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99>.