



THE UNIVERSITY OF QUEENSLAND
School of Information Technology and Electrical Engineering

**Network Packet Generation Tool for Testing
a Network Intrusion Detection System**

A thesis submitted for the degree of
Master of Science (Computer Science)

by
Johnson Fong

Supervised by
Dr Peter Sutton

1 June 2007

The Head
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, QLD 4068

Dear Professor Paul Bailes,

In accordance with the requirements of the degree of Master of Science (Computer Science), I present the following thesis entitled "Packet Generation Tool for Testing a Network Intrusion Detection System". This work was performed under the supervision of Dr Peter Sutton.

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text and bibliography, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Yours sincerely,

Johnson Fong

To God

Acknowledgements

This thesis would not have been completed without the support, guidance and encouragement of many people. Thus, I personally wish to thank all of them, whether they are named here or not.

First, I must express my sincere appreciation to my supervisor, *Dr Peter Sutton*, who has guided me throughout my project with his invaluable advice and patience.

I also would like to thank *Satwant Sandhu*, a PhD student who is working in the same field as me for his technical support in computer-related matters.

I gratefully thank *Glen Reeve* and *Henry Ly* for their generous friendship and support, especially during the final stages of preparing this thesis.

Last but not least, I wish thank to my dearest parents, to whom I have dedicated this thesis.

Johnson Fong
1 June 2007

Abstract

Network Intrusion Detection System (NIDS) is a system designed to detect the evidence of computer network intrusion. It resides on a computer connected to a segment of a network and monitors network traffic on that network segment, looking for indications of ongoing or successful attack.

Snort is an example of an open source software implementation of a NIDS, and there is a hardware implementation of NIDS currently being developed at The University of Queensland.

Research and experience have shown that no NIDS is perfect; it may miss attacks (false negative) or report false alarms for legitimate traffic (false positive). This project aims to develop a highly configurable and flexible testing tool that is able to generate different streams of network packets based on set of parameters a user provides.

The goal of this project is to utilise the testing tool to perform an analysis and experiment on the software and hardware implementation of the NIDS. The test results produced by the NIDS hardware are expected to be better than the software implementation in terms of performance, whilst also sustaining the same correctness as the NIDS software.

A testing tool has been developed which can analyse Snort rules and automatically generate a series of alert-triggering traffic that conforms to the rules. Furthermore, based on parameters the user specifies into the tool, a combination of alert-free and alert-triggering network traffic can be generated in an attempt to emulate a more realistic workload for the NIDS performance evaluation. One could fine-tune the parameters further to reflect the needs of producing a more suitable and effective test.

A formal software evaluation was conducted and the tool was found to be successful. The tool is close to completion and meets the project scope. It is capable of handling most aspects of rule options, including the Perl compatible regular expression and flow option, which are only available to recent versions of Snort (2.0.0 onwards). Since the software development follows an incremental model Software Development Life Cycle, an incremental testing strategy was adopted so that early feedback can be provided to the developer.

Table of Contents

Acknowledgements	iv
Abstract	v
List of Figures	ix
List of Tables.....	x
List of Tables.....	x
1. Introduction	1
1.1 Topic.....	1
1.1.1 Introduction to NIDS	1
1.1.2 Introduction to attack signatures and Snort rules ...	2
1.2 Project Scope.....	4
1.3 Aim	4
1.4 Goal.....	5
1.5 Statement of Purpose	5
1.6 Contribution of the Thesis.....	5
1.7 Report Outline.....	6
2. Review of background and relevant work.....	7
2.1 Relevant Work on NIDS Testing	7
2.1.1 Robustness testing	8
2.1.2 Performance testing	13
2.2 Relevant work on NIDS Evasion Techniques	16
2.2.1 Insufficiency of information on the wire.....	16
3. Software Specification Requirement.....	19
3.1 Software Requirement.....	19
3.2 Use Case Specifications	19
3.2.1 Generate Normal Traffic	19
3.2.2 Generate Bad Traffic	20
3.2.3 Simulate Intrusion.....	22
4. Design and Implementation	23
4.1 Selection of Environment.....	23
4.2 Selection of Traffic Generator	23
4.3 Overall System Architecture	25
4.4 Command Line Arguments.....	27
4.5 Generate Normal Traffic.....	28
4.6 Generate Bad Traffic	33

4.6.1	Rule Header	34
4.6.2	Rule Body	35
4.7	Intrusion Simulation	43
4.7.1	Portscan.....	43
4.7.2	Fragmentation	46
4.8	Summary	47
5.	Evaluation	48
5.1	Generate Normal Traffic.....	49
5.2	Generate Bad Traffic	51
5.3	Simulate Intrusion	55
5.4	Comparison with specification	57
5.5	Validation	58
5.6	Comparisons with related product.....	58
5.7	Original Contribution.....	59
5.8	Self-Evaluation	59
5.9	Summary	60
6.	Future Development	61
6.1	Generate Normal Traffic.....	61
6.2	Generate Bad Traffic	62
6.3	Simulate Intrusion	62
7.	Conclusion.....	63
	References	65
	Appendix A: Grammar rule for Snort Signature	68
	Appendix B: Configuration File Used for Testing.....	71
	Appendix C: Rule File (local.rules) Used for Testing.....	73
	Appendix D: Supported PCRE.....	77

List of Figures

Figure 1: Snort being placed in a network [12].	2
Figure 2: A Snort Rule Header [3]	3
Figure 3: Agent Architecture [5]	10
Figure 4: NextGen Architecture [10]	14
Figure 5 TTL-based Insertion Attack [17]	17
Figure 6: Generate normal traffic use case diagram	20
Figure 7: Generate bad traffic use case diagram	21
Figure 8: Simulate intrusion use case diagram	22
Figure 9: NIDS testing Topology	25
Figure 10: TestingTool Architecture.	25
Figure 11: A vanilla TCP scan result when a port is open [25]	45
Figure 12: A vanilla TCP scan result when a port is closed [25]	45
Figure 13: An inverse TCP scan result when a port is open	45
Figure 14: An inverse TCP scan result when a port is closed	45

List of Tables

Table 1: Snort Bugs Found by AGENT [4].....	11
Table 2: Evaluation Results for Snort [7]	12
Table 3: Common Ambiguities Identified in a Network [4]	18
Table 4: Generate normal traffic use case	20
Table 5. Generate bad traffic use case	21
Table 6: Simulate intrusion use case.....	22
Table 7: TestingTool command line arguments.....	27
Table 8: Summary of the PCRE evaluation results.....	53
Table 9: Comparison of Previous Work with TestingTool	58

1.Introduction

This chapter will provide a brief introduction to the topic of this thesis, as well as the major aims and goals of the project with their relevance to the topic.

1.1 Topic

The topic of this project is a creation of a packet generation tool that is capable of performance testing a Network Intrusion Detection System (NIDS).

1.1.1 Introduction to NIDS

A NIDS is a system that constantly monitors network traffic for malicious activity. It captures network packets through a passive network interface, operating in promiscuous mode, reading all traffic off the wire regardless of its destination. As packets are captured by a NIDS, they are stored in a kernel buffer, until an application reads them out and examines them against a database of attack signatures [4].

Once a packet is matched against an attack signature, an intrusion is detected. A NIDS has a number of preconfigured actions from which the user should choose **one** to handle intrusion. The most common action is to send alerts to an administrator. These alerts may be in the form of outgoing email, pop-up windows or even automated phone calls depending on the severity of the perceived intrusion. An action could also just be logging the packet without generating an alert [3].

It is important to understand that the NIDS will not block the bad traffic of the intrusion; NIDS will allow the bad traffic through. The NIDS is similar to burglar alarms in the physical world: both of which are passive responses [1]. When an intrusion is detected, these can only alert and cannot block the traffic; whereas an IPS (Intrusion Prevention System) with an active response could terminate traffic that is seen as being malicious. However, the IPS is out of the scope of this project.

Figure 1 below shows where Snort, a widely deployed example of NIDS, is typically placed on a network, sniffing off packets that cross its path and listening to all traffic going to the end system.

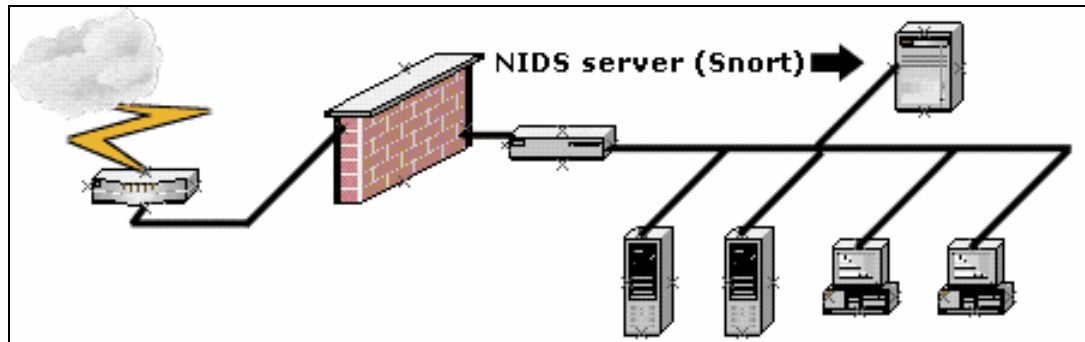


Figure 1: Example of where a Snort is typically placed in a network, sniffing off packets that cross its path and listening to all traffic going to the end system [12].

1.1.2 Introduction to attack signatures and Snort rules

An attack signature is a pattern or characteristic describing what an attack looks like. The signatures are incorporated into the Snort ruleset, which is used to detect intrusions by Snort.

The ruleset consists of two parts, a rule header and a rule body, as shown in Figure 2. An incoming packet is first tested against a Snort header rule. If the Snort header rule is matched, the incoming packet is then tested against a set of signatures in the rule body, including a set of payload signatures for content matching against the payload of the incoming packet [10].

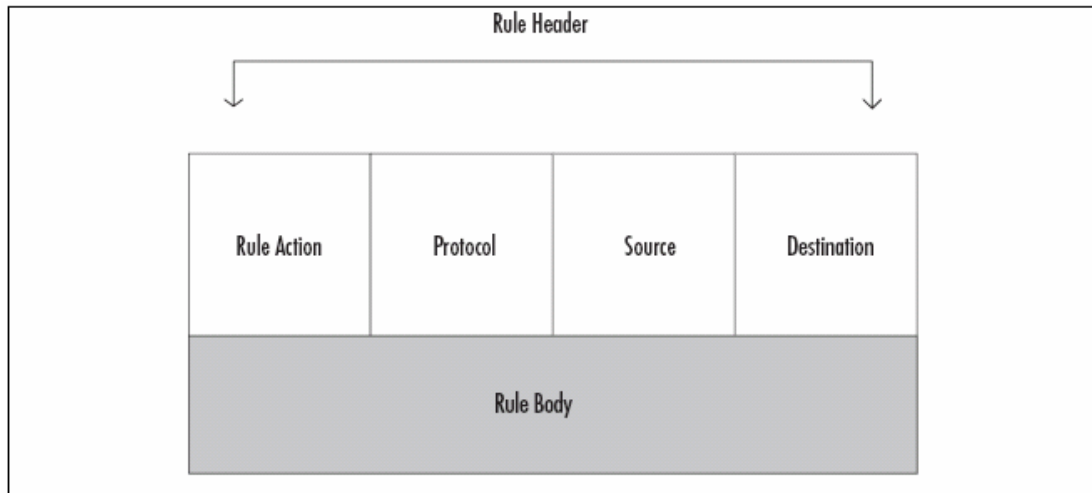


Figure 2: A Snort Rule Header [3]

The following is an example of a Snort rule for triggering an alert on any TCP packet destined for the Telnet service with the word "administrator" in the payload of the packet [3]:

```
alert tcp any any -> any 23 (content: "administrator";
nocase;)
```

A rule is made up of two parts, a rule header "alert tcp any any -> any 23" and a rule body "(content: "administrator"; nocase;)".

The rule header is the main portion of the signature. It defines the action (`alert`) to be taken when the rule is flagged, and identifies packet-level information, such as which protocol is in use (`tcp`), the source (`any`) and destination ports (`23`), and finally the source and destination IP addresses (`any`).

Snort rules do not require the body field to be complete rule definitions. The body is to extend the breadth of the rule definition beyond simply alerting, based on packet source and destination. Regarding the example given above, the body is used to test the payload of a packet containing the word "administrator", disregarding the text case of the word.

1.2 Project Scope

The main focus of this project is to develop a program that parses in Snort rulesets and generates streams of packets capable of triggering the rules. The program should also allow users to specify a range of parameters for configuring types of packets that will be generated, such as TCP, UDP and IP.

The pseudo streams of packets generated are fired against the NIDS, which may trigger alerts and produce a number of different testing results. The results could then be gathered together to form statistics and graphed to see the relationships between some of the parameters against the performance of the NIDS.

The purpose of the parameters is an attempt to exercise different internal functions of the NIDS for measuring the performance and robustness in some areas under certain conditions. The internal functions are for example the TCP engine, the HTTP decoder and the pattern matching mechanism.

The NIDS employed for this project is called Snort [3], the de facto standard for intrusion detection in the industry. The program is developed in C, implemented on top of an open source network packet injection utility called Nemesis.

One thing to note here is that conducting the actual testing of the NIDS and evaluating its performance are beyond the project scope, since the development of the testing tool is the purpose of this project.

1.3 Aim

The aim of the project is to develop a highly configurable and flexible testing tool that is able to generate different streams of network packets based on the rule sets and the parameters which the user provides.

1.4 Goal

The fundamental goal of the project is to utilize the testing tool to perform an experiment on the NIDS software, and then apply the same experiment with the same stream of packets to the NIDS hardware. The results are then compared with each other for evaluation. The test results produced by the NIDS hardware are expected to be better than the software implementation in terms of performance, whilst also **sustaining the same correctness** as the NIDS software.

1.5 Statement of Purpose

Evaluating and testing the performance of the Network Intrusion Detection System has been found to be challenging, because different systems have different operational environments and may employ a variety of techniques for generating alerts corresponding to attacks [1, 2]. The importance of this project is that by inspecting the outcomes of the testing, it is possible to:

- To gain quantitative insights concerning the capabilities and limitations of Snort.
- To gain a better understanding of the inner working components: how components function and interoperate with each other internally.
- To allow the comparison and evaluation of the two different implementations of the Snort, quantifying the performance and correctness accurately.

1.6 Contribution of the Thesis

Testing is an activity that is required as part of a project development process. It can assure the quality of the Snort hardware that is currently under development as a PhD thesis. Since quality is a major issue in engineering projects, lack of quality in mission critical applications, such as intrusion detection systems, can have a major impact on the safety of a network.

1.7 Report Outline

The remainder of the thesis is structured as follows:

Chapter 2 presents a review of the background and relevant work in the field of the load testing of the NIDS and evasion techniques.

Chapter 3 refines the overall scope of the thesis into software specific requirements for the testing tool.

Chapter 4 presents the testing tool that has been developed. The design and implementation of the testing tool is described.

Chapter 5 presents a detailed evaluation of the software implementation.

Chapter 6 presents some future research possibilities that can extend this work.

Chapter 7 states the overall conclusions of the thesis.

2. Review of background and relevant work

This chapter covers the background material and work relevant to the project. It places this project within a broader context in the field of NIDS testing and NIDS evasion methods. In section 2.1, Relevant Work on NIDS Testing, there is a discussion on the work carried out previously on NIDS testing by various research projects, comparing and contrasting their approaches.

There is one point that is common to all the work: the fundamental techniques for generating exploits or attacks that are used as test cases for NIDS testing. The techniques for generating test cases all fall under the same principles described by *Ptacek and Newsham* [4]. They used **semantics preserving** IP and TCP attack transformations to elude all NIDS they tested, based on the **inherent ambiguities** of a network. By semantics preserving, they mean that an attack transformation will not alter the semantics of the attack.

Thus, in section 2.2, Evasion Methods, *Ptacek and Newsham's* work [4] is investigated regarding the **inherent weaknesses** of NIDS and techniques for NIDS evasion, such as fragments manipulation and Denial of Service.

2.1 Relevant Work on NIDS Testing

There are two parts to this section 2.1. In part 2.1.1, Robustness testing, a summary of tools for robustness testing is firstly presented, followed by a brief overview of two models presented in [5] and [7], which are being adopted for robustness testing of NIDS. In part 2.1.2, Performance testing, an overview of a model presented in [10] for performance testing NIDS is provided.

2.1.1 Robustness testing

In the past few years, the majority of research into the area of intrusion detection is mainly focused on robustness testing, penetration testing and vulnerability research. There are numerous tools that already exist to perform all sorts of tricks and clever techniques attempting to break NIDS in various ways. These tools are often call IDS stimulators, for example AGENT [5], Mucus [16], Snot [5], Stick [8], Fragroute [13], Whisker [15] and IDSwakeup [9].

Summary of IDS stimulators

Mucus is a network traffic generator, developed at the University of California in 2003. It is very similar to Snot and Stick in that it uses Snort rules as input and generates a series of packets that conform to the input rules. Its goal is to force a NIDS to generate a large number of detection alerts (alert storm) in order to desensitize the NIDS administrator and hide attacks in the network traffic [20]. The implementation of the traffic generation code is adapted from Snot, which parses in one rule at a time and generates a single (stateless) TCP/UDP packet each time, randomizing payload where possible [5].

Opposite to the single packet generation in Mucus, Fragroute [13] and Whisker [15] are capable of generating full-session TCP-based attacks. Fragroute implemented most of the attacks described in *Ptacek and Newsham* [4]. The techniques will be briefly explained in Section 2.2, *Evasion Methods*. The techniques include testing of the timeouts on NIDS, checking reassembly of fragmented packets (overwriting the same data with different content), simulating delays and packet loss in network programs, and randomization of IP parameters to evade operating system fingerprinting (i.e. some older OS may still accept a packet with a bad IP checksum). Fragroute turned these attack theories into practice, although most of the attacks have been addressed since Snort version 1.9 [3] has been produced.

Whisker is another one of the few tools that is capable of generating full-session TCP-based attacks. It is a URL scanner, which is used to search for known vulnerable CGIs on websites. Whisker can also generate HTTP attacks, mutate an HTTP request so much that the NIDS will get confused, but the web server will still be able to understand it. There is an enormous number of ways to write a URL differently while preserving the same semantic. Here is an example of an attack employing the Self-referenced directory technique:

```
"GET cgi-bin./././phf? HTTP/1.0"
```

Because this attack string would not be interpreted by the NIDS in the same way as "GET cgi-bin/phf? HTTP/1.0", the attack will pass the NIDS undetected.

Another common technique for NIDS **obfuscation** is URL Encoding; there are multiple ways to encode each letter in a URL. For example, Hex encoding ("A" becomes "%41"), UTF-8 encoding and Microsoft %U encoding etc. Whisker is the first tool that has implemented these types of NIDS evasion techniques. While it is officially deprecated in 2003, many tools such as Nikto [15] and Nessus [21] make use of Whisker's library that encompasses all these evasion techniques.

An overview of two robustness testing models

The article *Automatic Generation and Analysis of NIDS Attacks* [5] provides a formal model for combining the attack transformations underlined in *Ptacek and Newsham* [4], as well as developing a tool that automatically generates transformed attacks that serve as test cases to evaluate Snort. Their testing tool is called AGENT (Attack Generation for NIDS Testing tool), which reads Snort rules and transforms them using the formal model for Attack Derivation for computing attack instances. The Attack Derivation Model is a mathematical model that is based on **natural deduction** [6] and can be used to define the black and white hat problems.

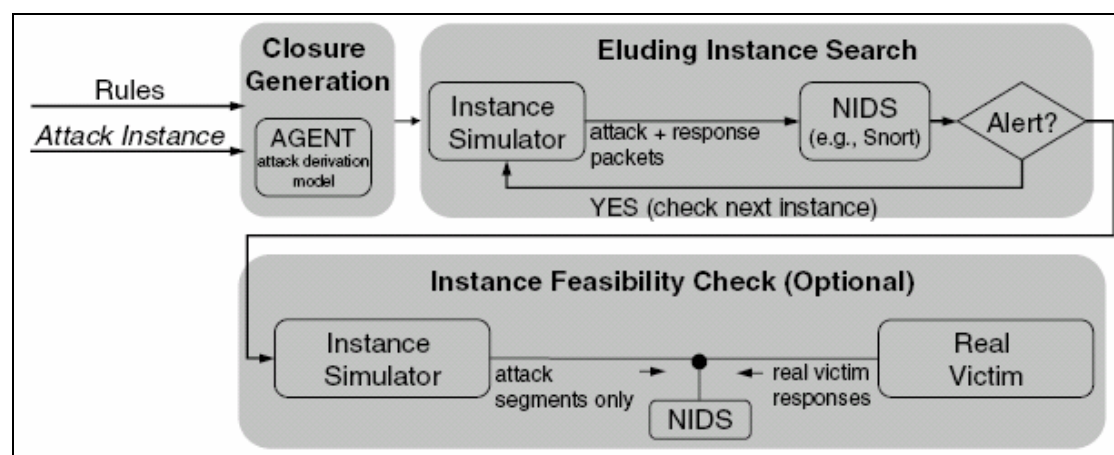


Figure 3: Agent Architecture [5]

The testing approach taken by AGENT is shown in Figure 3. Initially, given a set of rules or a typical attack instance (i.e. FTP buffer overflow attack "CWD ~root"), AGENT computes a root of the attack instance, and generates many different instances derived from this root using the attack derivation model mentioned above. Then having computed all these different instances, AGENT finds an attack instance that eludes Snort during the Eluding Instance Search stage.

The attack transformation techniques used to generate these instances are mentioned in *Ptacek and Newsham* [4]. Examples of the attack transformation techniques are *HTTP URL Encode*, *IP Fragmentation*, *TCP Retransmission* and *FTP Padding* [5]. These techniques are based on the **inherent weaknesses** of NIDS also described in [4], which are briefly discussed in section 2.2, *Evasion*

Methods.

During this testing, when Snort missed an instance, the experiment was stopped to find out the reason for this. Table 1 presents a summary of the vulnerabilities their testing effort exposed. Evasive RST exposed vulnerability in a Snort mechanism that tracks the state of a TCP session. FTP padding exposed vulnerability in the interaction between a Snort mechanism that handles TCP packets and the one that performs pattern matching.

Name	Description	Implications: it is possible to eludes Snort for any
Evasive RST	Snort accepts an out-of-window TCP RST packet, and stops tracking a live TCP connection.	TCP-based attack. Fixed in Snort v2.0.2.
Flushing ¹	Snort misses a signature that is fragmented over several TCP packets.	attack whose signature can be inflated by an application-level rule.
HTTP space padding	Exploits Snort's default configuration together with its nature to report only a single alert per TCP packet. Snort misses the attack or generates a general alert instead of the <i>perl-in-cgi</i> alert.	Web-CGI attack. With a default configuration, Snort misses the attack; with a user-defined configuration, Snort generates a general HTTP alert rather than the specific alert for the attack.
HTTP multiple requests ¹	Snort analyzes only a single HTTP request per TCP packet.	Web-CGI attack. Fixed in Snort v2.1.0.
FTP Padding	Snort does not recognize a certain type of regular expressions.	attack with a signature such as "foo*bar". Fixed in Snort v2.0.6.

¹ This vulnerability was reported at the time of the experiment.

Table 1: Snort Bugs Found by AGENT [4].

In summary of the testing performed by AGENT, the **robustness** of Snort was being tested: its vulnerabilities were investigated for systems exploitation and the percentage of the attack instances that successfully eluded Snort was measured. The testing outcomes demonstrated that Snort remained vulnerable to varieties of attacks.

The testing approach taken by AGENT is fairly similar when **compared** to the one in the article *Testing Network-based Intrusion Detection Signatures Using Mutant Exploits* [7]. This article [7] has a similar testing technique, which is based on a mechanism that generates a large number of variations of an exploit by applying **Mutant operators** to an exploit template.

In a way, this is the same as AGENT applying the Attack Derivation Model to a Snort rule or attack instance. They both have the same idea of eluding a signature-based NIDS by transforming an attack instance to avoid its payload being matched to the NIDS signature. The NIDS testing approach taken by Mutant [7] was that, initially, given a common exploit (i.e. ftp CWD ~root), mutation techniques are applied to the exploit, and generate from a minimum of about a hundred mutations to a maximum of several hundred thousand different attacks. These mutated attacks are called **Mutant Exploits**.

The mutation techniques used to generate the Mutant exploits are also mentioned by *Ptacek and Newsham* [4]. Examples of the mutation techniques are *IP Packet Splitting*, *FTP Evasion Techniques*, *HTTP Evasion Techniques* and *Alternate Encodings*. These techniques are based on the **inherent weaknesses** of NIDS also described in *Ptacek and Newsham* [4], which are briefly discussed in Section 2.2, *Evasion Methods*.

During the testing, the testing process was stopped when a mutant that was able to evade detection was found. Table 2 shows that Snort correctly detected all instances of baseline attacks. The exploit mutation engine, however, was able to automatically generate mutated exploits that evaded Snort’s detection engine for six out of the ten attacks [7].

Exploit	Baseline attack	Mutated Attack	Evasion Techniques
WU-ftp Remote Format String Stack Overwrite	Detected	<i>Evaded</i>	Telnet control sequences Shellcode mutation IP packet splitting
WU-imap Remote Buffer Overflow	Detected	<i>Evaded</i>	Prepend zeros to numbers Shellcode mutation
IIS Escaped Characters Double Decoding	Detected	Detected	
Microsoft DCOM-RPC	Detected	Detected	
IIS Extended Unicode Directory Traversal	Detected	<i>Evaded</i>	URL encoding
NSISlog.DLL Remote Buffer Overflow	Detected	Detected	
IIS 5.0 .printer ISAPI Extension Buffer Overflow	Detected	Detected	
WS-FTP Server STAT Buffer Overflow	Detected	<i>Evaded</i>	Telnet control sequences IP packet splitting
OpenSSL SSLv2 Client Master Key Overflow	Detected	<i>Evaded</i>	SSL NULL record insertion
Apache HTTP Chunked Encoding Overflow	Detected	<i>Evaded</i>	HTTP protocol evasion

Table 2: Evaluation Results for Snort [7]

In summary of the testing performed by Mutant, a similar approach to AGENT testing is adopted. The testing of Mutant also aimed at validating and testing the robustness of Snort. A similar observation that Snort remained vulnerable to variations of exploits based on these mutation techniques was made.

2.1.2 Performance testing

As opposed to the robustness testing approach carried out in previous research projects, the approach taken here is more towards the idea of NIDS evaluation and **performance** testing. The experiment for this study focuses on providing some useful indications and statistics about average performances of NIDS under tests.

The intention of this study is to perform a complete evaluation, exercising as many functions and characteristics of NIDS as possible, which is the main purpose of the tool being developed. The user is able to configure the tool by specifying a range of different parameters, so that the user can identify correlations between different parameters and the testing outcomes; for example, the number of false positive alerts and the behaviour of NIDS when run with different proportions of bad traffic.

Although looking at the testing approach from the point of view of robustness and validation of NIDS is reasonable as well, coming up with ways of breaking the NIDS is not the primary concern here. The research article that is most related to this approach is *Generating Realistic Workloads for Network Intrusion Detection Systems* [10]. A workload model was developed that provides accurate estimates compared to real workloads. The model attempts to emulate a synthetic traffic mix of different applications, reflecting characteristics of each application and the way these interact with Snort. A tool called *NextGen* has also been implemented, which adapts to the workload model, simulating application traffic for measuring the **capacity** of Snort and testing its performance.

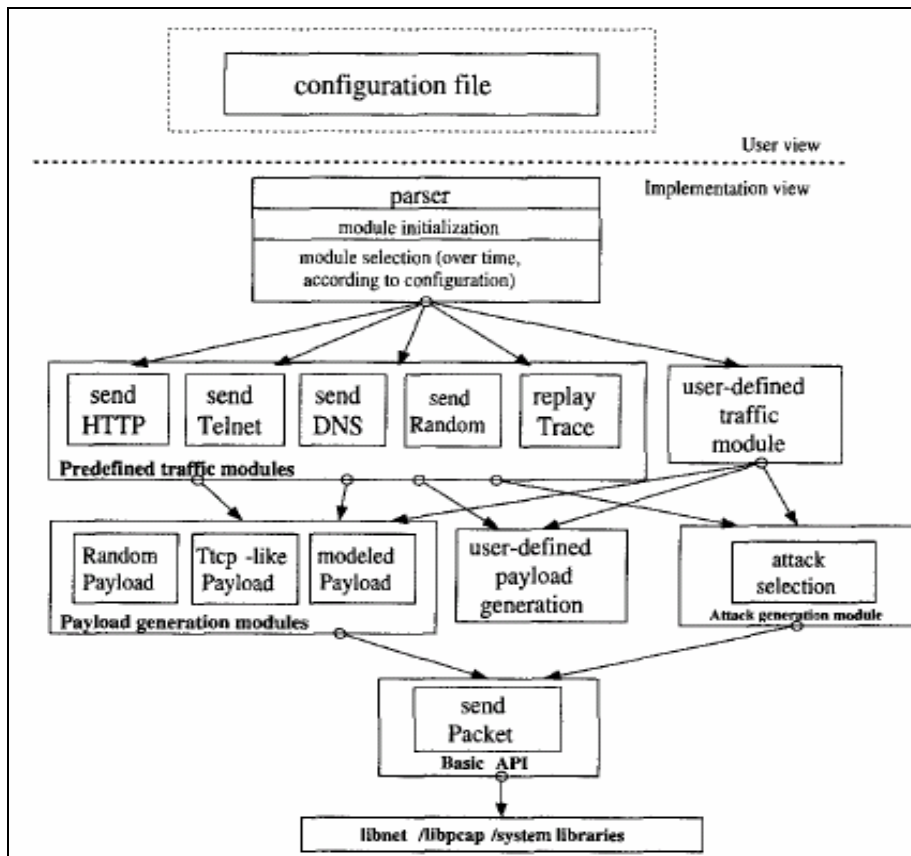


Figure 4: NextGen Architecture [10]

The performance testing method taken by NextGen was to firstly utilize modules emulating a mix of different protocols as part of a synthetic workload. Each module shown in Figure 4 is responsible for generating a specific application protocol, such as HTTP or DNS queries. The *payload generation modules* are used to construct payloads that contain certain types of attacks against Snort.

NextGen measures the capacity of Snort by generating a synthetic workload based on the data from a variety of network traces from different environments, with Snort running with all 1,923 available rules activated. The capacity of Snort was measured with different pattern matching algorithms, such as E²xB and MWM, on a gigabit testbed. The experiment shows that for both the E²xB and MWM algorithm, the capacity of Snort reached around 210Mbit/s and 36ms. These values agree with the measured capacity for one GB of normal traffic.

The outcome of the performance test is in terms of **Mbit/s** capacity and the processing time is in **milliseconds**, which is different to the robustness testing from previous projects that evaluated Snort in terms of whether an attack has successfully evaded Snort.

As well as measuring the performance, NextGen can also measure the robustness of Snort against overload attacks, by feeding in regular (attack-free) packet traces through NextGen. It will then transform regular payloads to payloads containing attack strings. The attack strings will result in the worst-case behaviour by a Snort pattern matching engine [10]. The overloading techniques are discussed in Section 2.2.2, *CPU exhaustion*.

By **contrasting** the NIDS testing approaches that are mentioned above, it can be seen that the performance testing methodology taken by NextGen is more appropriate to this project than the robustness testing and NIDS validation methodology taken by AGENT and Mutant. Thus, the work done by NextGen is the one that is made use of in this study. A discussion of the testing approach taken by this project is detailed in Chapter 3, Project Plan.

2.2 Relevant work on NIDS Evasion

Techniques

There are two inherent weaknesses of NIDS mentioned in *Ptacek and Newsham* [4]:

1. Insufficiency of Information on the Wire
2. Inherently Vulnerable to Denial of Service attack

Ptacek and Newsham also mention *Insertion* and *Evasion* techniques as being a common method for exploiting these weaknesses.

Only *Insufficiency of Information on the Wire* will be discussed here with a review of the *Insertion* technique, as these are highly relevant to this project's aim; their pioneering work has made a significant impact on the NIDS community, which will influence the direction of the project and decisions made throughout its duration.

2.2.1 Insufficiency of information on the wire

This weakness is based on the **ambiguities** of a network, which causes inconsistency between the NIDS and the end-systems that it watches. Inconsistency means that NIDS and end-systems can potentially see different streams of packets and/or process packets in an inconsistent manner with each other.

The following is an example of one of the ambiguities of a network:

NIDS may not be certain as to whether the TTL field in the IP packet is large enough for the number of hops to the destination [4].

This ambiguity can cause the inconsistency that NIDS may receive a different stream of packets to an end-system that it watches.

Figure 5 presents a simple scenario where an attack can exploit this ambiguity using an **insertion** attack to defeat a signature analysis, allowing it to slip malicious packets past NIDS [17].

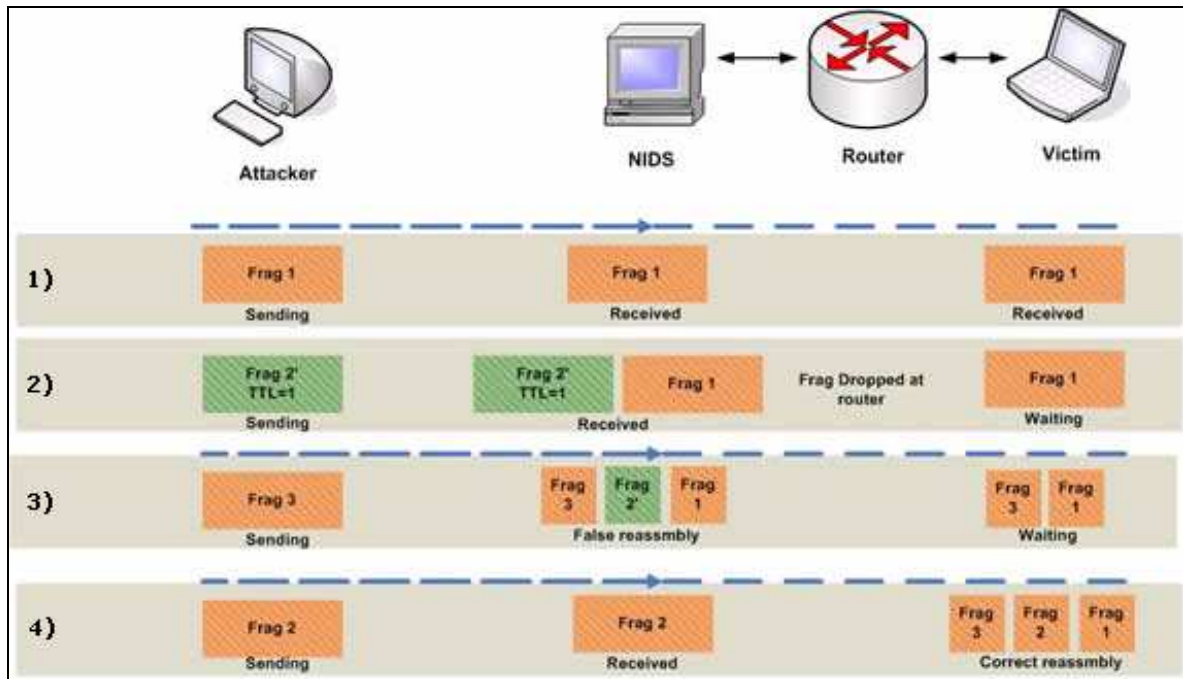


Figure 5 TTL-based Insertion Attack [17]

- 1) An attacker splits an attack message into three fragments and sends the first fragment with a TTL value large enough to reach the NIDS and the victim.
- 2) The attacker sends a fake second fragment 2' with a TTL value of one, which can only be received by the NIDS and not the victim.
- 3) The attacker sends the third fragment with a valid TTL. The NIDS will then perform a TCP-reassembly on fragments (1, 2', 3), whereas the victim still waits for the genuine second fragment.
- 4) At this stage, the attack will now send the genuine second fragment with a valid TTL. This makes the victim perform a reassembly on fragments (1, 2, 3) and receive the attack message [17].

In this example, due to the ambiguity of the IP TTL field, the NIDS sees a different stream of packets to the victim which made the network vulnerable to the Insertion attack. Insertion occurs when NIDS sees more packets than the end system it watches.

- The NIDS sees: frag1-frag2"-frag3-frag2
- The victim sees: frag1-frag2-frag3 (attack)

A second source of information about the topology of the network is required to resolve this TTL field ambiguity. This weakness is called **Insufficiency of Information on the Wire**, because a packet alone is not enough for NIDS to predict accurately whether a given machine on the network is even going to see a packet, let alone process it in the expected manner [4].

The following Table 3 is a summary of the most common ambiguities of a network. By removing these ambiguities, most of the vulnerabilities found in Snort can be eliminated. Handlet [18] and Shankar [19] present techniques that remove TCP and IP ambiguities from network connections. These techniques can be used to prevent vulnerabilities, for example, the *Evasive RST* vulnerability mentioned in Section 2.1 that is found in Snort by AGENT.

Info Needed	Ambiguity
Network Topology	IP TTL field may not be large enough for the number of hops to the destination
Network Topology	Packet may be too large for a downstream link to handle without fragmentation
Destination Configuration	Destination may be configured to drop source-routed packets
Destination OS	Destination may time partially received fragments out differently depending on its OS
Destination OS	Destination may reassemble overlapping fragments differently depending on its OS
Destination OS	Destination host may not accept TCP packets bearing certain options
Destination OS	Destination may implement PAWS and silently drop packets with old timestamps
Destination OS	Destination may resolve conflicting TCP segments differently depending on its OS
Destination OS	Destination may not check sequence numbers on RST messages

Table 3: Common Ambiguities Identified in a Network [4]

3. Software Specification

Requirement

This chapter specifies all the requirements pertaining to the development of the NIDS Testing Tool.

3.1 Software Requirement

The testing tool requires a PC with a GCC compiler and Nemesis 1.4 to be installed on Linux kernel 2.6. When this research was conducted, Snort 2.6.1.1 was the most current version.

3.2 Use Case Specifications

Use Case Specification is a common technique for capturing software requirements in software engineering, detailing the requirements of the system in terms of how it is to be used. The major functions of the NIDS testing tool is to generate normal (alert-free) traffic, generate bad (alert-triggering) traffic and simulate intrusion. Their functional requirements are captured in the following use case diagrams.

3.2.1 Generate Normal Traffic

The use case diagram and use case to generate normal traffic is depicted in Figure 6 and Table 4 respectively.

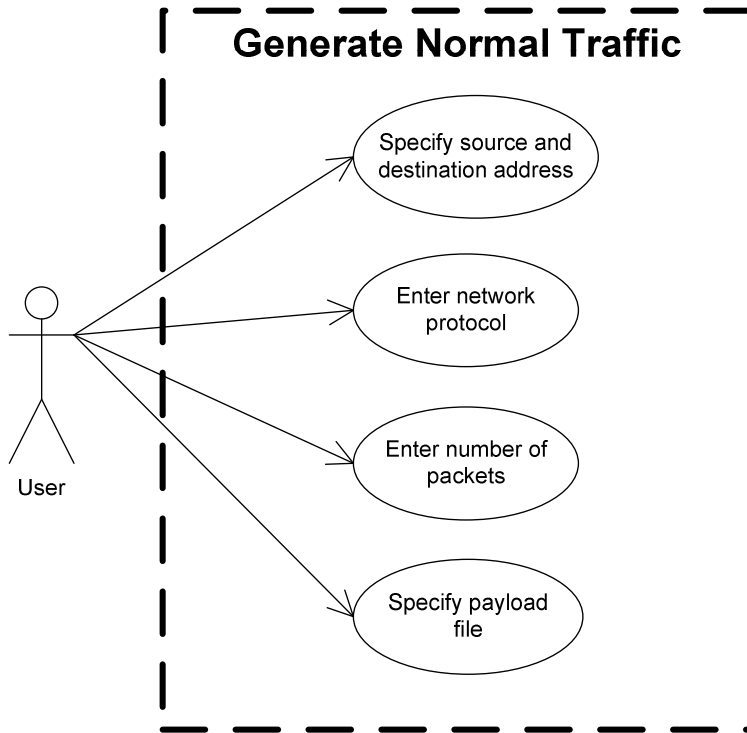


Figure 6: Generate normal traffic use case diagram

Use Case	Generate normal traffic
Brief Description	A user generates normal alert-free packets
Actors	User
Main Flow of Events	<ol style="list-style-type: none"> 1. The user has to first specify the source and destination IP address. 2. The user will enter the network protocol. 3. The user will specify a number between 1 – 100 000 for the number of packets they wish to generate. 4. The user will attach a payload onto each packet by specifying the location of the payload file.
Alternative Flow of Events	The user will not specify a payload.
Post Condition	Traffic with the specified protocol is sent to the destination.

Table 4: Generate normal traffic use case

3.2.2 Generate Bad Traffic

The use case diagram and use case to generate bad traffic is depicted in Figure 7 and Table 5 respectively.

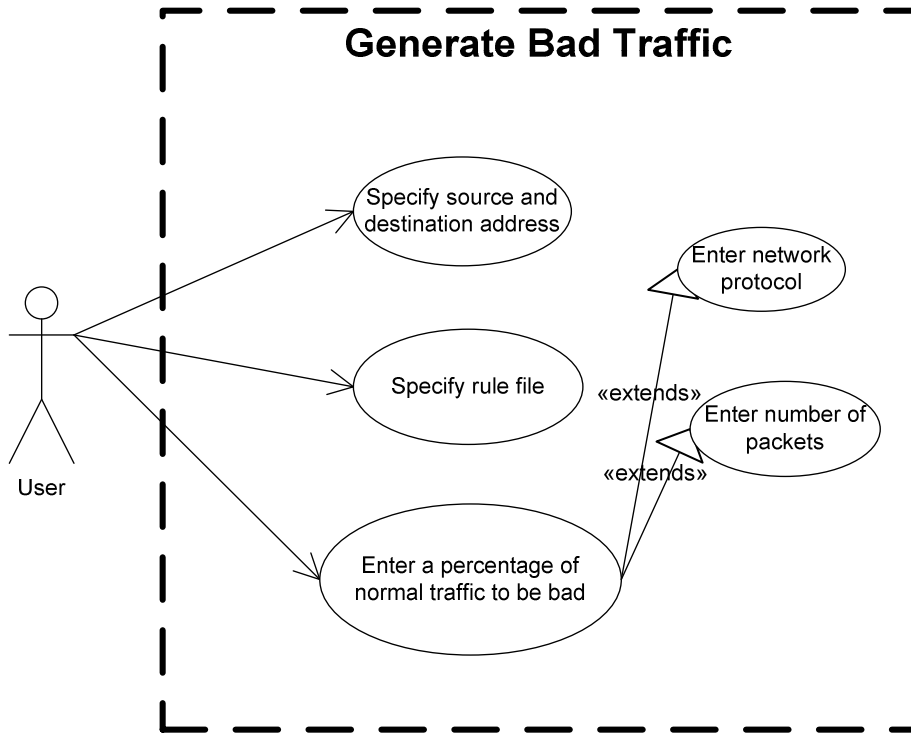


Figure 7: Generate bad traffic use case diagram

Use Case	Generate bad traffic
Brief Description	A user generates alert-triggering packets
Actors	User
Main Flow of Events	<ol style="list-style-type: none"> 1. The user first has to specify the source and destination address. 2. The user will specify a location of a rule file 3. The user will enter a network protocol. 4. The user will enter the number of packets to generate. 5. The user will specify a percentage of the packets to be bad. 6. The system will parse rules from the rule file that have the same protocol as entered by the user, and will translate them into bad packets.
Alternative Flow of Events	<p>The user will not specify a percentage of packets to be bad.</p> <p>The system will parse all the rules in the rule file and translate them into the corresponding bad packets.</p>
Post Condition	Rules in the rule file will be translated into bad packets, according to the command line and configuration file.

Table 5. Generate bad traffic use case

3.2.3 Simulate Intrusion

The use case diagram and use case to simulate intrusion is depicted in Figure 8 and Table 6 respectively.

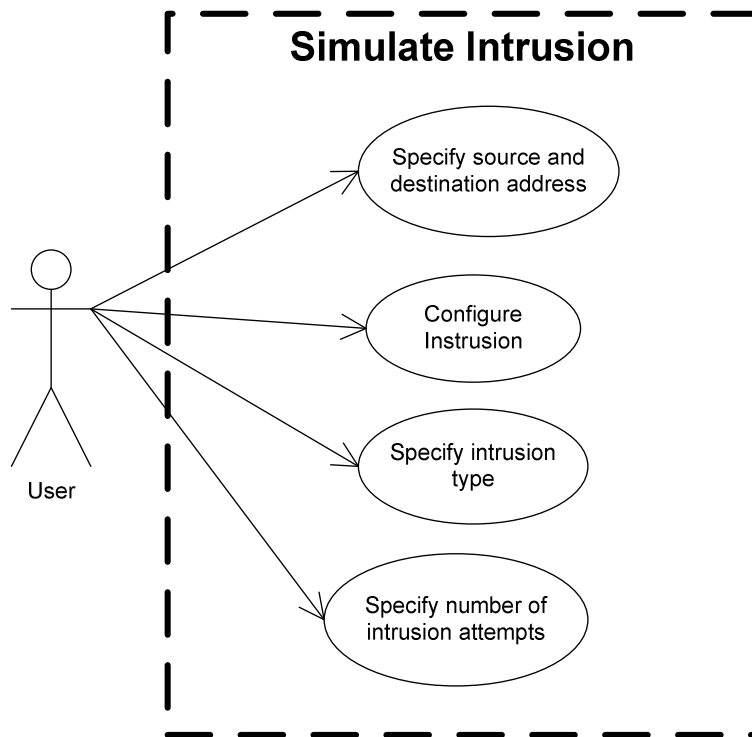


Figure 8: Simulate intrusion use case diagram

Use Case	Simulate Intrusion
Brief Description	A user simulates different types of intrusion
Actors	User
Main Flow of Events	<ol style="list-style-type: none"> 1. The user first has to specify a source and destination address. 2. The user will configure the intrusion setting before simulation. 3. Once complete, the user has to enter a type of intrusion to simulate. 4. The user will repeat the intrusion simulation by specifying a number of attempts.
Alternative Flow of Events	The user will not specify a number of intrusion attempts. The system will only simulate the intrusion once.
Post Condition	A simulation of the type of intrusion specified occurs.

Table 6: Simulate intrusion use case

4.Design and Implementation

This chapter presents TestingTool, a complete and flexible application that has been designed to test the performance of Snort. The requirements mentioned in Chapter 3 have been incorporated in TestingTool. TestingTool comprises a front-end parser and a back-end traffic generator. The parser is responsible for translating Snort rules and user parameters into a scripting file. The traffic generator is responsible for executing the script against a target.

4.1 Selection of Environment

Two languages were initially considered for the development of TestingTool, Java and C. Java has the advantage of providing extensive support for system portability, but the disadvantage is the fact that Java compiles its code into non-native executables which is typically slower than an equivalent program written in C. A fundamental requirement is that TestingTool must be fast in order to generate enough traffic into the network in order to stress test Snort. C was chosen as the language of choice, even though Java has more extensive abstraction of data structures, which would have reduced development time. Another benefit of using C is that it also has a richer networking API than Java. Everything in TestingTool, from the parser to the traffic generator, is written in C.

4.2 Selection of Traffic Generator

With today's networking environment, it is not surprising that there is a lot of open source traffic generator software programs freely available and already tested such as Hping, Sendip and Nemesis. There is only the requirement of selecting a traffic generator for the purposes of this study. Hping, Sendip and Nemesis are command-line driven, suitable for automation and scripting. They are written in C for Unix-like operating systems. They are fast and efficient, designed for testing firewalls and NIDS [23].

As with Snort 2.6.1.1, the protocols that Snort only analyses for suspicious behaviour are TCP, IP, UDP and ICMP. Nemesis is currently capable of generating other packet types such as ARP, IGRP, OSPF, RIP and DNS. It will be readily adaptable if, in future, Snort supports these protocols. Sendip and Hping, on the other hand, do not support as many protocols and their command line syntaxes are more complicated, so creating packets is more difficult and error-prone than Nemesis. With the desirable properties of Nemesis, it is the preferred tool to be employed for testing Network Intrusion Detection Systems. This is a strong example of software reuse from the software engineering perspective.

TestingTool uses Nemesis to generate packets onto a network. The following is an example of a Nemesis command for injecting a TCP segment onto a network:

```
Nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -fSA -s  
111111 -a 222222 -x 3333 -y4444
```

This command will inject one packet containing a TCP header (TCP), a source IP address (-S 192.168.0.11), a destination IP address (-D 10.1.1.1), SYN and ACK flag (-fSA), a sequence number (-s 111111), an acknowledgement number (-a 222222), a source port number (-x 3333) and a destination port number (-y 4444).

To generate network traffic, TestingTool crafts Nemesis commands into a scripting file, and Nemesis executes the script to inject packets onto the network.

4.3 Overall System Architecture

An overview of the NIDS testing topology is depicted in figure 9 below.

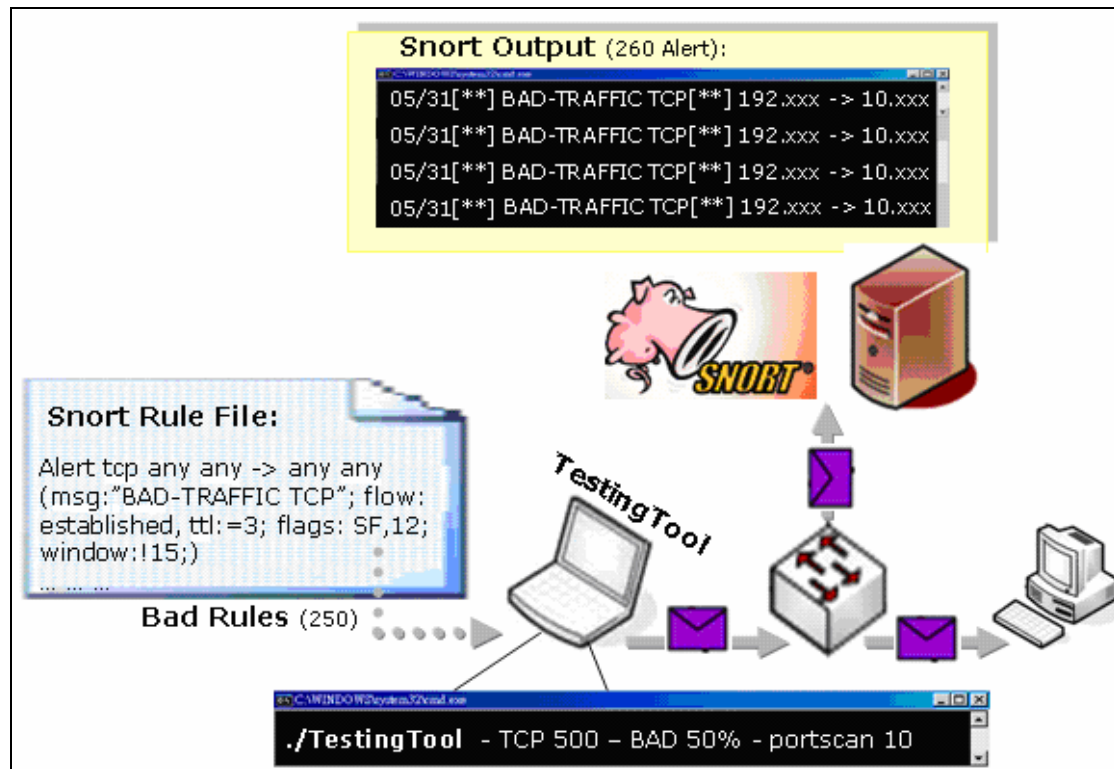


Figure 9: NIDS testing Topology

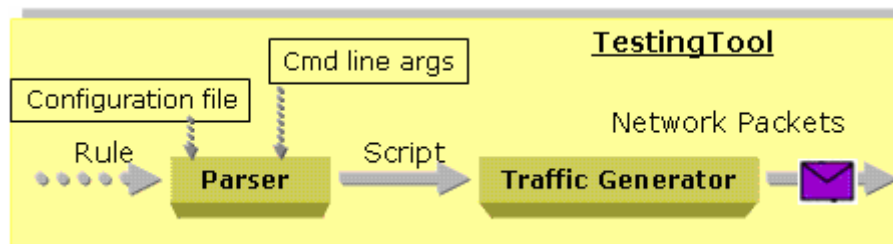


Figure 10: TestingTool Architecture.

Figure 10 shows the logical relationships between the front-end parser and the back-end traffic generator within TestingTool. One of the requirements for this tool is to be a highly configurable and flexible. One possible design is to have a large number of command line arguments to allow the user to specify various parameters into the tool. However, that design is not practical, as too many

arguments can cause confusion. The approach taken here is slightly different; it was decided that a combination of a configuration file as well as command line arguments would be the most suitable. TestingTool will load in the configuration file before the generation of any traffic.

The whole process can be briefly summarized as follows:

- Setup Phase (done manually)
 1. The user creates a file (rule file) to hold a selection of exploit rules.
 2. The user then specifies the location of the rule file into the configuration file.
 3. The user sets the network options in the configuration file.
 4. The user sets the execution options in the command line arguments.

- Execution Phase (done automatically)
 1. TestingTool reads in the command line and configuration file to see what type of packets should be built.
 2. According to the configuration, TestingTool may parse in a rule file, translating the rules inside into Nemesis commands.
 3. The Nemesis commands are placed in a scripting file.
 4. The traffic generator (Nemesis) executes the script against the target.
 5. Snort captures the packets generated and raises alerts for any signature matches.

4.4 Command Line Arguments

Command Argument	Usage example	Description
Generate Normal Traffic		
-TCP number_of_packet	./TestingTool -TCP 10	Generate 10 normal TCP segments
-UDP number_of_packet	./TestingTool -UDP 10	Generate 10 normal UDP datagrams
-IP number_of_packet	./TestingTool -IP 10	Generate 10 normal IP packets
-ICMP number_of_packet	./TestingTool -ICMP 10	Generate 10 normal ICMP packets
-Stream number_of_packet	./TestingTool -Stream 10	Generate 10 TCP streams
Generate Bad Traffic		
[Generate Normal Traffic] -BAD %_of_bad_Traffic	./TestingTool -TCP 10 -BAD 50%	Generate 10 TCP segments, 50% is bad
Simulate Intrusion		
-Frag number_of_attempts	./TestingTool -Frag 2	Simulate 2 attempts of fragmentation attack
-Portscan number_of_attempts	./TestingTool -Portscan 2	Simulate 2 attempts of portscan events

Table 7: TestingTool command line arguments

Table 7 is a summary of the command line arguments for TestingTool. The command line is designed to be highly flexible. Arguments to generate normal traffic can be mixed together to generate traffic that has a combination of protocols.

For example:

```
./TestingTool -TCP 10 -UDP 10 -IP 10 -ICMP 10
```

This generates 10 TCP, 10 UDP, 10 IP and 10 ICMP packets.

The command line is designed to allow a percentage of bad traffic to be specified on each of the protocols.

For example:

```
./TestingTool -TCP 10 -Bad 50% -UDP 10 -IP 10 -BAD 20%
```

In this case, 10 TCP packets are generated and 50% of them are bad, 10 UDP packets are generated, 10 IP packets are generated and 50% of them are bad.

The command line is also designed to allow a mixture of intrusion attempts, normal and bad traffic to be generated at the same time.

For example:

```
./TestingTool -TCP 10 -Bad 50% -UDP 40 -Portscan 2  
-Frag 2
```

In this case, 10 TCP packets are generated and 50% of them are bad, 40 UDP packets are generated, 2 portscan events and 2 fragmentation attacks are simulated.

4.5 Generate Normal Traffic

This section explains how alert-free traffic is generated onto a network. The purpose is to simulate application traffic for measuring the capacity of Snort and testing its performance. The design goal of TestingTool is that it has to be fast in order to stress Snort's capacity. Since Snort has to pattern match the payload against the signatures on every packet it inspects, the capacity of Snort is determined by observing how much throughput Snort can handle before it starts dropping packets. TestingTool generates normal traffic by crafting Nemesis commands according to the protocol and number of packets the user has specified as per the command line arguments.

The following is an example of a command line for generating 100 normal TCP segments:

```
./TestingTool -TCP 100
```

TestingTool is designed to craft TCP, IP, UDP and ICMP Nemesis commands, since these are the only protocols Snort can support.

The following are examples of alert-free Nemesis commands for each protocol:

TCP Segment (Total size travelled on the wire is 60 bytes excluding payload. TCP header takes 20 bytes. IP header takes 20 bytes. Ethernet header takes 14 bytes)

```
nemesis TCP -S 192.168.0.11 -D 10.1.1.1 -fA -s  
423423 -a 432423 -x 4444 -y 12345 -p payload
```

UDP Datagram (Total size travelled on the wire is 60 bytes excluding payload. UDP header takes 8 bytes).

```
nemesis udp -S 192.168.0.11 -D 10.1.1.1 -x 4444 -y  
12345 -p payload
```

IP Packet (Total size travelled on the wire is 60 bytes excluding payload. IP header takes 20 bytes.)

```
nemesis IP -S 192.168.0.11 -D 10.1.1.1 -p payload
```

ICMP Packet (Total size travelled on the wire is 60 bytes excluding payload. ICMP header takes 8 bytes)

```
nemesis ICMP -S 192.168.0.11 -D 10.1.1.1 -p payload
```

Alert-free Nemesis commands were intended to be crafted as simply as possible, avoiding any options where possible. It is important that alert-free traffic will not trigger any false positive.

Before any commands are crafted, a configuration file has to be set:

IP Addresses

The source (-S) and destination (-D) addresses are specified in the configuration file and not on the command line, since they do not change very often. The configuration file contains address variables `EXTERNAL_NET` which represents the source address (i.e. `-S 192.168.0.11`) and `HOME_NET` which represents the destination address (i.e. `-D 10.1.1.1`).

The configuration file supports address variables to be defined by a straight numeric IP address and a Classless Inter-Domain Routing (CIDR) block [22]. For example, the address/CIDR combination 192.168.1.0/24 would signify the block of addresses from 192.168.1.1 to 192.168.1.255.

If the variable EXTERNAL_NET is being defined with this designation, TestingTool would randomly pick any address in that range as the source address. The CIDR designation gives a convenient and flexible method of designating large address spaces.

The configuration file supports address variables to be defined by a keyword "any", which means that TestingTool will randomly pick any IP addresses for address variables. The configuration file also supports address variables to be defined by a negation operator "!" that can be applied to the IP address and CIDR block.

Source Port Number

The source port number (-x) has been hard-coded to 4444, due to the fact that BSD uses ports 1024 to 4999 as ephemeral ports. An ephemeral port is assigned to a client of a client-server communication. It is temporary and therefore insignificant in this context.

Destination Port Number

The destination port number of a generated packet can be specified in the configuration file. A variable PORT is declared in the file. It is used to define a destination port number ranging from 1- 65535. The port number will be incremented by one for every additional packet generated and starts again from port 1 after port 65535. This is an attempt to simulate realistic traffic instead of having all traffic assigned to the same IP and port. The port number zero is not used due to the fact that Snort raises an alert for traffic using port zero since it is a reversed port.

Payload

The payload of a packet is specified in the configuration file. A variable PAYLOAD is declared in the file. It is defined with the location of a file. Its size must be no more than 65415 bytes, since the IP Total Length (TL) field is 16bits, thus a maximum IP packet is only 64kB.

Flag

For all TCP segments generated, the ACK flag (-fA) is hard coded as almost all real TCP traffic has the ACK flag being set.

Sequence and Acknowledgement Number

The sequence (-s) and acknowledgement (-a) numbers is a 32-bit number that is randomly generated.

TestingTool is also capable of crafting TCP stream Nemesis commands consisting of a three-way handshake; a SYN, a SYN-ACK and an ACK packet. For TCP stream traffic, the sequence and acknowledgement numbers are generated according the following set of Nemesis command syntax:

TCP Stream (Total size travelled on the wire is 180 bytes excluding payload)

```
nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -fS -s "X" -x
4444 -y 12345
nemesis tcp -S 10.1.1.1 -D 192.168.0.11 -fSA -s "Y" -a
"X+1" -x 12345 -y 4444
nemesis tcp -S 192.168.0.11 -D 10.1.1.1"dest" -fA -s
"X+1" -a "Y+1" -x 4444 -y 12345 -P payload
```

Where:

"X" = initial sequence number, "X" will begin with 1.

"X+1" = 2.

"Y" = acknowledge number, "Y" will begin with 3

"Y+1" = 4

If two TCP streams are generated, the following will be the value for each variable on the 2nd set of Nemesis commands:

"X" = 5

"X+1" = 6

"Y" = 7

"Y+1" = 8

So basically:

"X" = 1,5,9,13,17,21... (If there are 1 million streams, there will be 1 million different "x", max value of x = 2^{32})

"X+1" = 2,6,10,14,18,22...

"Y" = 3,7,11,15,19,23...

"Y+1" = 4,8,12,16,20,24...

This process can ensure that each TCP segment in a stream will have a unique sequence and acknowledgement number that do not overlap with other TCP segments.

4.6 Generate Bad Traffic

TestingTool generates bad (alert-triggering) traffic by parsing in a Snort rule file and automatically translating the Snort rules inside the rule file into the Nemesis commands. The commands are then executed by the traffic generator to generate bad traffic. The purpose is to exercise the signature matching engine and content matching within Snort, since Snort needs to check every packet against hundreds of signatures and almost all of them involve deep payload inspection. This section examines the problems encountered with Snort signature parsing and how they are addressed in TestingTool. Five Snort rule examples are presented to explain how different Snort rules are turned into bad traffic.

The following is an example of a command line for generating alert-triggering packets:

```
./TestingTool -TCP 10 -BAD 50%
```

In this example, TestingTool generates 10 TCP packets and 50% (5) are alert-triggering TCP packets. A parser within TestingTool has to parse in a rule file; its location is specified in the configuration file under a variable name called `BAD`. In order to generate 5 TCP bad packets, the parser translates the first 5 TCP rules from the rule file into the corresponding Nemesis commands.

If there were only 1 TCP rule in the rule file on which the bad packets are generated from, this simply means that the same TCP rule is being repeatedly translated multiple times until 5 TCP rules has been translated in total.

Prior to the design and implementation of the TestingTool parser, the possibility of using Snot [5] and Mucus [16] to parse the Snort signatures was explored. The problem with Snot and Mucus is that it has been designed to recognise the Snort signature language used in Snort version 1.8.3 and 1.8.6 respectively; however, the intent of this work was to focus on the latest version of Snort at this time of writing. The most significant differences between the version 2.6 and 1.8 are two additional options, flow and PCRE. Over 80% of the

default Snort rules contain these two options. This fact was the primary motivator in the decision to design and implement a more capable parser.

Parsing of individual Snort rules is handled by the `add_new_rule()` function. It starts with parsing the header portion of a rule (action, source and destination information, protocol). If a '(' token is encountered in the input stream, it indicates a presence of options and forces the `add_new_rule()` function to enter a loop which terminates when a final ')' is met. Or if an unrecoverable error has occurred, an error message is printed and the application is terminated.

For every loop iteration, a `process_option()` function is called. It slips down a recursion path for options parsing. The `process_options()` function obtains an option name and finds out whether the option in question is required for a Nemesis command generator. This process can be unified by providing a table of option names vs. function addresses to handle these options. When an option is identified as required, the proper option validity flag is set and the option data is then read and processed until the end of the option. An option ends when the ';' token is found.

For the grammar rule that TestingTool uses to parse the Snort rule, please refer to Appendix A: Grammar Rule for Snort.

4.6.1 Rule Header

TestingTool is capable of parsing a rule without a body, since Snort rules do not require the body field to complete the rule definition.

1. Rule example 1- Rule header

This is an example of a Snort rule defined only with a rule header:

```
alert tcp $EXTERNAL_NET 2222 -> $HOME_NET !3333
```

The following is the corresponding generated Nemesis command that will trigger the rule example:

```
nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -x 2222 -y
1111
```

The `EXTERNAL_NET` and `HOME_NET` address variable in the Snort rule are translated to the source (`-S 192.168.0.11`) and destination (`-D 10.1.1.1`) IP address respectively in the Nemesis command. These two IP addresses are defined in the configuration file under the two address variable names of `EXTERNAL_NET` and `HOME_NET`.

The IP address defined in the configuration file can be a CIDR address block (i.e. `192.168.0.0/24`), allowing the user to simulate more realistic traffic that is coming from a distributed source address. The address is generated as a random value and the following formula is then used to bring the random address into the masked address space:

$$\text{address} = (\text{random_address} \& \sim\text{mask}) | \text{base_address}$$

The negated address (i.e. `!192.168.1.0/24`) uses a slightly different approach where random IP values are generated and checked if a new value is valid and does not conform to the given rule mask. It also takes into account TCP ports (i.e. `!3333`) and sequence numbers. Output Nemesis commands are generated by a `generate_output()` function.

4.6.2 Rule Body

The body of a Snort rule is important for identifying complex attack sequences [22]. The rule body contains many different options. TestingTool is designed to support the following rule options.

Payload option:

`Uricontent`, `content`, `distance` and `PCRE` (Perl Compatible Regular Expression).

Non-payload option

`flow`, `ttl`, `tos`, `id`, `flags`, `seq`, `ack`, `window`, `itype`, `icode`, `icmp_id`, `icmp_seq` and `sameip`

During the development of the parser, the rule options that occurred more frequently in the standard Snort ruleset were implemented at an early stage. Options that occurred less frequently in the Snort ruleset, for example, `fragoffset`, were implemented later. The rule options that are deprecated, for example, `content-list` were not implemented. The `flow` and `PCRE` options were implemented first, as they are the most common features used.

2. Rule example 2-Nonpayload options

This rule example from a rule file contains the non-payload options `ttl`, `tos`, `id`, `flags`, `seq`, `ack` and `window`:

```
alert tcp 10.1.1.1 2222 -> 192.168.0.4 3333 (ttl:<5;
tos:>6; id:!323; flags:SF,12; seq:5612; ack:3974;
window:128;)
```

The following is the corresponding generated Nemesis command that will trigger rule example 2:

```
Nemesis tcp -S 10.1.1.1 -D 192.168.0.4 -T 3 -t 8 -I
200 -fSFCE -s 5612 -a 3974 -w 128 -x 2222 -y 3333
```

All rule options are translated directly onto the Nemesis command. For example, `ttl <5` is translated to `-T 3`, `tos >6` is translated to `-t 8`, `id !323` is translated to `-I 200`. In order to support relational usage such as `<` `>` signs, `Range` type was added to the implementation, which encapsulates range start and end values. For the above `tos>6` as an example, the range would be constructed as: `start=7` and `end=255`. A random value is selected from the valid range.

3. Rule example 3 - flow established, from client option

This rule example from a rule file contains non-payload options `flow established`, `from client`:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (flow:
established,from_client; ttl:44; )
```

The following is the four corresponding generated Nemesis commands that will trigger rule example 3, assuming `$HOME_NET` is defined as `10.1.1.1` and `$EXTERNAL_NET` is defined as `192.168.2.7` in the configuration file:

```
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -fS -s
4293129348 -x 41569 -y 60637
nemesis tcp -S 192.168.2.7 -D 10.1.1.1 -fAS -s
4223664093 -a 4293129349 -x 60637 -y 41569
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -fA -s
4293129349 -a 4223664094 -x 41569 -y 60637
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -T 44 -fA -s
4293129349 -a 4223664094 -x 41569 -y 60637
```

The `flow` option allows rules to only apply to certain directions of the traffic flow. This allows rules to only apply to clients or servers, and the rule examples 3, 4 and 5 were chosen to be clients. This allows packets related to `$HOME_NET` clients viewing web pages to be distinguished from servers running the `$HOME_NET` [22].

The `established` keyword on the Snort rule means an established TCP connection consists of `SYN`, `SYN/ACK` and `ACK`. The three-way handshake is required before Snort starts logging packets for any other rule options. That is why only the fourth Nemesis command contains the `TTL` rule option which is transmitted after the three-way handshake has been established.

Instead of generating four Nemesis commands as mentioned above to trigger rule example 3, an alternative design is to combine the third and the fourth Nemesis commands together, so now only three Nemesis commands are required to be generated to trigger rule example 3. For example:

```
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -fS -s
4293129348 -x 41569 -y 60637
nemesis tcp -S 192.168.2.7 -D 10.1.1.1 -fAS -s
4223664093 -a 4293129349 -x 60637 -y 41569
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -T 44 -fA -s
4293129349 -a 4223664094 -x 41569 -y 60637
```

The reason why combining the third and fourth Nemesis commands are still able to trigger rule example 3 is due to the fact that, before the introduction to the Flow option, the older version of Snort only looks for the SYN/ACK flag (`flags: A+`) to determine established TCP connections [22].

Since both sets of commands will do the same thing, it was decided to use the second set of Nemesis commands with three packets to trigger rule example 3, as fewer packets means that it is less error-prone and less likely to produce false positives.

4. Rule example 4 – flow established, to_client option

This rule example from a rule file contains non-payload options

```
flow established, to_client
alert tcp $HOME_NET any -> $EXTERNAL_NET any (flow:
established,to_client; ttl:44;)
```

The following is the corresponding generated Nemesis commands that will trigger rule example 4, assuming `$HOME_NET` is defined as 10.1.1.1 and `$EXTERNAL_NET` is defined as 192.168.2.7 in the configuration file:

```
nemesis tcp -S 192.168.2.7 -D 10.1.1.1 -fS -s
1002262987 -x 32048 -y 40387
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -fAS -s
1940626795 -a 1002262988 -x 40387 -y 32048
nemesis tcp -S 192.168.2.7 -D 10.1.1.1 -fA -s
1002262988 -a 1940626796 -x 32048 -y 40387
nemesis tcp -S 10.1.1.1 -D 192.168.2.7 -T 44 -fA -s
1940626796 -a 1002262989 -x 40387 -y 32048
```

One thing to note here is that there is a subtle difference between rule example 3 (`flow: established,from_client;`) and rule example 4 (`flow: established,to_client;`).

A TCP connection has to be established from server to client instead of client to server from the previous rule example 3. The three-way handshake is initiated by the server coming from `EXTERNAL_NET` to

HOME_NET. The client may only begin communication after the TCP connection has been established. Note the fourth Nemesis command containing the option TTL value 44 (-T 44) is only transmitted after the connection is established to the client. The Nemesis commands cannot be combined, unlike rule example 3, since they contain different destination addresses.

5. Rule example 5 - payload options

This rule example from a rule file contains payload options Uricontent, content, distance and PCRE:

```
alert tcp 10.1.1.1 any -> 192.168.2.5 $HTTP_PORTS
(msg:"SPYWARE-PUT Hijacker comet systems runtime
detection - update requests";
flow:from_client,established; uricontent:"v="; nocase;
uricontent:"t="; nocase; uricontent:"c="; nocase;
content:"Host|3A|"; nocase;
content:"update.cc.cometsystems.com"; distance:0;
nocase;
pcre:"/\x2F[^\r\n]*\.(dat)|(xml)\?[^\r\n]*v=[^\r\n]*t=
[^\r\n]*c=/Ui";
pcre:"/^Host\x3A[^\r\n]*update\.cc\.cometsystems\.com/
smi"; classtype:misc-activity; sid:5831; rev:1;)
```

The following is the corresponding generated Nemesis commands that will trigger rule example 5:

```
nemesis tcp -S 10.1.1.1 -D 192.168.2.5 -fS -s
4225227072 -x 2195 -y 80
nemesis tcp -S 192.168.2.5 -D 10.1.1.1 -fAS -s
2147473338 -a 4225227073 -x 80 -y 2195
nemesis tcp -S 10.1.1.1 -D 192.168.2.5 -fA -s
4225227073 -a 2147473339 -x 2195 -y 80 -P 5831.sid
```

The payload file, 5831.sid, contains:

```
Host:
2update.cc.cometsystems.comxml?:<Q.i"CgUs:p`GWQk)|v=bt
b/H[_llt=_]m2BWicu@0gc=v=t=c=Host:update.cc.cometsyste
ms.com
```

The crafted Nemesis commands consist of a three-way handshake. The payload is appended to the last Nemesis command with the same reason as explained for rule example 3. TestingTool parses each of the payload options (`content`, `PCRE` and `uricontent`) to create a payload file that has the same name as the `sid` value. The `sid` value is used to uniquely identify Snort rules. This information allows the created payload file to identify the corresponding rules easily.

TestingTool parses the `PCRE` option and the parsing output is always placed at the beginning of a payload file. The output of the `content` and `uricontent` are placed after the `PCRE` output. This is due to the fact that some `PCRE` options require the matching pattern to appear at the start of a line, where no `content` and no `uricontent` option have such restriction.

Note that multiple `PCRE` options can be specified in one rule. This allows rules to be tailored for less false positives. When there are two `PCRE` options in a rule, the `PCRE` option with this pattern, `/^`, is placed at the front, such as the payload for rule example 5:

```
Host:
2update.cc.cometsystems.comxml?:<Q.i"CgUs:p`GWQk)|v=bt
b/H[_1lt=_]m2BWicu@0gc=v=t=c=Host:update.cc.cometsyste
ms.com
```

With the colour coding as follows:

```
1stPCRE 2ndPCRE uricontent content
```

PCRE(Perl Compatible Regular Expression)

This section describes how a `PCRE` payload is generated with a given `PCRE` pattern.

```
pcre: "/^Host\x3A(?!\\n)\\s[^\r\n]*update\\.cc\\.cometsystems\\
.com/smi";
```

1. These characters "Host:" is placed at the start of a line.

```
pcre: "/^Host\x3A(?!\\n)\\s[^\r\n]*update\\.cc\\.cometsystems\\  
.com/smi";
```

2. This is a negative look-ahead. "Host:" is followed by a whitespace that is not a newline (i.e. space or tab), since the C language defines whitespace to be space, horizontal tab, newline, vertical tab, and form-feed etc.

```
pcre: "/^Host\x3A(?!\\n)\\s[^\r\n]*update\\.cc\\.cometsystems\\  
.com/smi";
```

3. A character that can be anything except carriage return is generated. Care has been taken to ensure the random data generated will not trigger other rules causing false positives. Whenever TestingTool encounters a negation (i.e. `^`) or a choice in a set (i.e. `[a-z]`), it will try to pick a random keyboard character (printable) first. If no keyboard character is available, it will then pick a random character from the set. If the character is followed by a repeat regular expression (i.e. `{100}`), it will reuse the character it has just picked instead of generating another new random character. It is unlikely for a rule to look for content with constant repeats of characters, such as "aaaaaa".

```
pcre: "/^Host\x3A(?!\\n)\\s[^\r\n]*update\\.cc\\.cometsystems\\  
.com/smi";
```

4. TestingTool includes newlines in the dot metacharacter and be case insensitive [22].

The parsing of PCRE is initially handled by a function `PCRE_compile()`, which is adapted from the PCRE library containing a set of native API and wrapper functions. `PCRE_compile()` accepts a string of regular expressions as input, and returns a PCRE data block which is a series of variable length tokens. In order to access each token, a `tokenize()` function creates an index over the stream of tokens and each token point to its position in PCRE data. Having an indexed list of tokens, a parse tree is created from these tokens with the following functions:

```
static Tree parse_regex( DList list, int *index ) ;
static Tree parse_branch( DList list, int *index ) ;
static Tree parse_brace( DList list, int *index ) ;
static Tree parse_zero( DList list, int *index ) ;
```

The tree is created of the following grammar:

```
regex -> branch { '|' branch }
branch -> { brace | zero | <token> }
brace -> ( BRA ) regex ( KET | KETR )
zero -> BRAZERO ( brace | branch )
```

BRA represents an open bracket. KET is a closed bracket. KETR represents a KET +. That is, this block will repeat at least once. BRAZERO on the other hand says that the next block is optional. Having created the parse tree, an `emit_tree()` function traverses the tree and gives the output accordingly.

Content

The `content` option in rule example 5 contains mixed text and binary data. The binary data is generally enclosed within the pipe (|) character and represented as bytecode (i.e. `content:"Host|3A|"`). When the `content` option has been parsed, the data is copied into the payload file `5831.sid`

4.7 Intrusion Simulation

This intrusion simulation is one of the features provided by TestingTool. It simulates intrusions that normally are not defined by rulesets or signatures. The purpose is to exercise different Protocol Anomaly Detection Engines (preprocessors), where Snort alerts packets that do not fit normal use of the packet's protocols. TestingTool is capable of simulating a Portscan intrusion and a Fragmentation attack.

4.7.1 Portscan

The Portscan feature simulates the process of identifying active network ports in the remote system. All attackers have to perform port scans on the target's machine before an attack can take place. The simulation of portscan is aimed to exercise a commonly used pre-processor called `sfportscan`, which is used to detect port scanning events. The design and implementation of the portscan simulation is adapted from a tool called `scanrand`[25].

The following is an example of a command line for simulating a portscan:

```
./TestingTool -portscan
```

The following is an example of the Nemesis commands which will trigger a portscan alert (total size travelled on the wire is 120 bytes):

```
nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -fS -s 1 -x  
4444 -y "port"  
nemesis tcp -S 10.1.1.1 -D 192.168.0.11 -fRA -s 11 -a  
2 -x "port" -y 4444
```

Port

The packets are sent to each port of the target host. The user can specify which port numbers to scan by defining a variable in the configuration file called `PORTSCAN` using dashes and commas. For example, `var PORTSCAN 10.0.1-255.1-10,20:80,137-139` works.

Port numbers to be scanned are stored on the single-linked list as pairs of start and end of the port range. This linked list approach provides flexibility for a virtually unlimited number of ports to be scanned.

Instead of using a comma/dash notated port range, several default port ranges have been precompiled into TestingTool. They are:

quick

Quick scanning hits the top one or two dozen TCP service ports that are often enabled on a given server. This translates directly to: 80, 443, 445, 53, 20-23, 25, 135, 139, 8080, 110, 111, 143, 1025, 5000, 465, 993, 31337, 79, 8010, 8000, 6667, 2049, 3306.

vquick

Very-Quick scanning hits the top five or six TCP service ports that are very often enabled on a given server. This translates directly to: 80,443,139,21,22,23

all

Scans all ports -- 1 to 65535.

Flag

For each port to be scanned, TestingTool generates two calls to the Nemesis application; one call to send TCP SYN packets to the specified remote system and the second call which sends RST/ACK packets back to the sender. This is a standard method of port scanning known as Vanilla SYN scanning.

Vanilla scanning works by sending a SYN packet to a port on the target machine and if the port is open, the target will reply with a SYN/ACK packet, as shown in Figure 11. If the port is closed, the target will reply with a RST/ACK packet, as shown in figure 12.

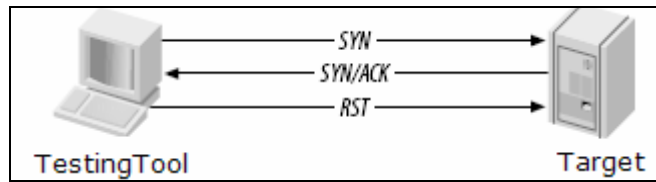


Figure 11: A vanilla TCP scan result when a port is open [25]

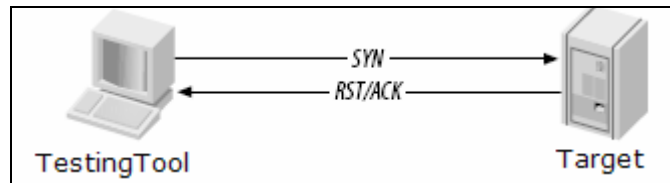


Figure 12: A vanilla TCP scan result when a port is closed [25]

An alternative approach is to perform a stealthy inverse TCP scan. This works by sending an XMAS probe with the `FIN/URG/PUSH` TCP flag set, as shown in figure 13. The RFC standard states that, if no response is seen from the target port, the port is open. This is stealthy, since sending garbage to each port usually will not be picked up. For all closed ports on the target host, `RST/ACK` packets are replied, as shown in Figure 14.

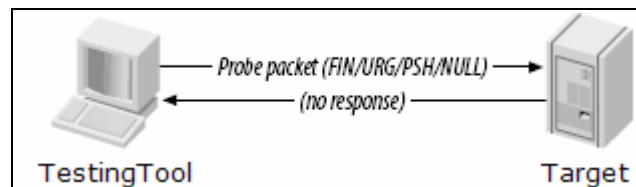


Figure 13: An inverse TCP scan result when a port is open

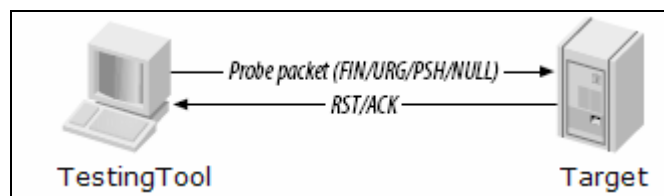


Figure 14: An inverse TCP scan result when a port is closed

It was decided not to implement the inverse TCP scan, since sending packets with the `FIN/URG/PUSH` TCP flag set has a higher probability of triggering other alerts (generating false positives) than simulating a vanilla scan, which only sends `SYN` and `RST/ACK` packet.

4.7.2 Fragmentation

The Fragmentation feature attempts to simulate fragments transmitted during an IP fragmentation attack, such as an overlapping fragment and a zero-byte fragment. The simulation of an IP fragmentation is aimed to exercise a sophisticated pre-processor called `frag3`, which is responsible for reassembling IP fragments as well as detecting a fragmentation attack. The design and implementation of the fragmentation simulation is inspired from a tool called `fragroute` [13].

The following is an example of a command line for simulating an IP fragmentation:

```
./TestingTool -frag
```

The following is an example of the Nemesis commands generated which will trigger three kinds of fragmentation alerts, `Fragmentation overlap` and `Zero-byte fragment packet`:

```
nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -I 243 -FM 8 -  
x 64667 -y 78973 -P tcp_payload  
nemesis tcp -S 192.168.0.11 -D 10.1.1.1 -I 243 -FM 8 -  
x 64667 -y 78973 -P tcp_payload  
nemesis ip -S 192.168.0.11 -D 10.1.1.1 -I 577 -FM 0 -x  
52424 -y 96761
```

The first two Nemesis commands are designed to be overlapping fragments, triggering the `Fragmentation overlap` alert, since they both contain the same fragment offset (`-FM 8`) and belong to the same IP packet with an ID value 243 (`-I 243`). Note that they also contain a 20 bytes TCP payload, where the third Nemesis command does not. This is because the third Nemesis command is designed to be a fragment containing only an IP header without any payload in order to trigger the `Zero-byte fragment packet` alert. An IP fragment usually contains a payload which is either an ICMP, TCP or UDP packet.

For flexibility reasons, these Nemesis commands can be modified; they are not precompiled into TestingTool. The Nemesis commands are generated from the parsing of several pre-defined Snort rules which are specified in a file called `frag.rules`. This allows the user to program their own fragmentation packets using Snort rules.

4.8 Summary

This chapter has explained the overall designs as well as the details of each major function of TestingTool.

Section 4.5 Generate Normal Traffic explained how TCP, IP, UDP and ICMP packets are being crafted into Nemesis commands.

Section 4.6 Generate Bad Traffic presented five examples of how Snort rules are turned into bad traffic. These Snort rules are: example 1 containing only a header, example 2 containing the non-payload options, example 3 containing the `flow: establishd, to_client; options`, example 4 containing the `flow: establishd, from_client; option`, and finally example 5 containing the payload options (PCRE, content and uricontent).

Section 4.7 Simulate Intrusion explained how actual intrusions such as portscan and fragmentation attack are being simulated.

Chapter 5 presents an evaluation of this system comparing it against both the specifications and related work. Chapter 6 details possible future improvements to TestingTool.

5.Evaluation

This chapter presents an evaluation of TestingTool consisting of testing the system with a few test cases and discussing the findings. TestingTool is also compared to other NIDS testing software and against the requirements outlined in Chapter 3. It is not possible to extensively test all features of TestingTool because of time and resource limitations. Therefore, in order to effectively evaluate the system, a testing strategy has been developed.

System testing was adopted for evaluating TestingTool, since it is the type of testing usually conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Black box testing of TestingTool is the preferred evaluation method to ensure that the program works as specified. White box testing was not done extensively due to the lack of time. Although white box testing would have been more thorough, the aim of this study is to produce a working application to test the NIDS performance and not be concerned about producing a robust secure application.

The three main areas that are focused on are Generate normal traffic, Generate bad traffic and Simulate Intrusion, since they are the three major requirements as defined in the Chapter 3 Software Specification. The evaluation of the each function was conducted using a rule file, configuration file and command line arguments as input to TestingTool. The generated traffic was then run against a Snort sensor. Please refer to Appendix B for the configuration file and Appendix C for the rule file (`local.rules`) used for testing. This experiment focused on providing useful indications about the effectiveness of the three major functions. It is not intended to evaluate the performance of Snort.

A number of observations were made during the testing of the system relating to the effectiveness of traffic generation. Presented below are test cases, results and observations for each function. Several issues (both positive and negative) that were identified are also discussed below. These findings indicate which parts of the

system should be improved in the future versions. Specific possible improvements are discussed in the Chapter 6.

5.1 Generate Normal Traffic

Planned Test Case

Test case 1: The user generates 10,000 concurrent TCP streams; each stream contains a payload of 1.2kB to ensure that TestingTool can generate a large amount of traffic without failing.

Test case 2: The user generates a mixture of TCP, UDP, IP and ICMP normal traffic with various numbers of packets for each protocol, and each packet contains a payload of 1.2kb. This is to ensure that a combination of protocols and different values are handled appropriately and no false positives arose due to the mix of normal traffic.

Recorded Test Results

The following table is the recorded test results. The description and reasons for testing are mentioned beneath each of the tests recorded.

#	Command line Inputs ./Testing	Expected Output	Actual Output	Pass /Fail
Generate Normal Traffic				
1	-Stream 10,000	10,000 TCP Streams generated	Results were expected	P
Test that TCP Stream Generation is functioning correctly				
2	-IP 10 -UDP 20 -ICMP 30 - TCP 40	10 IP, 20 UDP and 30 ICMP, 40 TCP Packet are generated	Results were expected	P
Test that combination of protocols and different values are handled appropriately				

Observation

Test case 1

Test case 1 took 23.2 seconds to complete on a low-end PIII 933 Mhz processor residing on a 100Mb test bed. The total size of the 10,000 TCP streams with a 1.2 kB payload being attached to each stream is 14,122,318 bytes (14MB), which was the amount that was transmitted over the network. The average transmission speed was 608,720.6 bytes/second or 4.86 Mbit/second. The speediness of normal traffic produced by TestingTool is crucial in determining the usefulness of the Generate Normal Traffic function. As previously stated in Chapter 4, the design goal of this function is to be fast in order to stress Snort's capacity. The performance of TestingTool was not up to expectation. An analysis was performed and it was discovered that the performance was due to a new process being spawned for each packet generated. The results could be improved by modifying Nemesis's source code to eliminate the start-up and tear-down overhead of spawning a new process. This will significantly reduce the processing time for each Nemesis command and would produce higher throughput. If more time had been permitted, this would be the more appropriate design.

Test case 2

Another factor in the evaluation of the Generate Normal Traffic function is how capable TestingTool can create a stream of packets with certain characteristics. Test case 2 demonstrated that TestingTool is able to generate normal traffic for TCP, UDP, IP and ICMP protocols with 10, 20, 30 and 40 packets respectively. It was observed that the generated traffic is good quality normal traffic, since no false positive alert (either from Snort rules or preprocessors) has arisen. In addition, it was observed that the source IP addresses on some packets are different to each other. This is because in the configuration file the EXTERNAL_NET address variable was specified as 192.168.2.0/24. Test case 2 demonstrated that TestingTool is capable of generating alert-free traffic with certain characteristics (i.e. various protocols, different source and destination addresses, different numbers of packets, each containing a fixed size payload, etc).

5.2 Generate Bad Traffic

The rule file being used in this evaluation can be found in Appendix C. The rule file is called `local.rules`, and contains 10 TCP, 10 UDP, 10 IP and 10 ICMP rules. The rule file contains all the rule options that were decided to implement in Chapter 4. They are:

```
flow, ttl, tos, id, flags, seq, ack, window, itype,
icode, icmp_id, icmp_seq, sameip, uricontent, content,
distance and PCRE.
```

The success of each rule translation was determined using the alerts generated by Snort and correlating them with the corresponding rule using the unique `sid` value on the alerts and rule.

Planned Test case

Test case 3: The user generates 44 TCP segments with 50% of them to be bad to ensure percentage of bad packets are generated correctly by TestingTool.

Test case 4: The user generates bad packets only to ensure all rules are being translated accurately.

Test case 5: The user generates multiple protocols of bad traffic to ensure multiple protocols in regards to % of bad traffic are generated correctly.

Recorded Test Results

The following table is the recorded test results. The description and reasons for testing are mentioned beneath each of the tests recorded.

#	Command line Inputs ./Testing	Expected Output	Actual Output	Pass /Fail
Generate Bad Traffic				
3	-TCP 88 -bad 50%	44 normal and 44 bad TCP packets are generated, causing 44 corresponding alerts to be outputted	Results were expected	P
Test that percentage of bad packets are being generated accordingly.				
4	-bad	All 40 Snort rules in a rule file are converted to bad packets, causing 40 corresponding alerts to be outputted	Results were expected	P
Test that all Snort rules in a rule file are translated correctly				
5	-IP 10 -bad 30% -UDP 20 -bad 50%	7 normal and 3 bad IP packets, and 10 normal and 10 bad UDP packets are generated, causing a total of 13 corresponding alerts to be outputted	5 normal and 5 bad IP packets, and 10 normal and 10 bad UDP packets are generated, causing 15 corresponding alerts to be outputted	F
Test that multiple protocols in regards to % of bad traffic are generated correctly.				

Observation

Test case 3

One of the requirements for TestingTool is to generate a mixture of normal and bad traffic, by having the user specify a percentage of the normal traffic to be bad. Test case 3 demonstrated that TestingTool is able to generate the correct amount of bad traffic according to the user input, generating exactly 44 bad TCP packets where the user specified 50% of 88 normal TCP packets.

There were only 10 TCP rules in `local.rules` on which the bad packets are generated from. 44 bad packets are generated; this simply means that the same TCP rules are being repeatedly translated multiple times. Each TCP rule is represented by at least 4 packets each.

Test case 4

Another factor in the evaluation of the Generate Bad Traffic function is how accurately TestingTool can translate Snort rules. Generate Bad Traffic is designed to parse in Snort rules that contain only the options that were decided to implement in Chapter 4. It is not sensible to use the default Snort rules to evaluate the effectiveness of Generate Bad Traffic, since the default Snort rule that accompanies Snort contains options that were not intended to be implemented.

The rules inside `local.rules` contain all the rule options stated in Chapter 4. Each rule is carefully designed for testing. Test case 4 shows that of the 40 total rules in `local.rules`, all of them were successfully translated by TestingTool. This test case demonstrates that TestingTool is able to reliably translate all the options that are in `local.rules`, except the Perl Compatible Regular Expressions (PCRE) option.

TestingTool can only translate PCRE options accurately to a certain degree. This is due to the fact that the PCRE language is highly extensive; PCRE options in some other rules may contain PCRE Extended Pattern which is currently not implemented by TestingTool. PCRE options can only be translated correctly with the absence of Extended Patterns. Please refer to Appendix D for a list of PCRE Extended Patterns. An experiment was conducted for evaluating the accuracy of PCRE translation using a default Snort rule `ftp.rules`, `web-client.rules` and `exploit.rules` instead of the one prepared in `local.rules`. This is due to the fact that the default Snort rules contain a more comprehensive and realistic set of PCRE.

Rule file	Rule contain PCRE	Correct Translated Rule
ftp.rules	77	27 (35.1%)
Web-client.rules	108	58 (53.7%)
Exploit.rules	31	8 (25.8%)
Total	216	93 (43.1%)

Table 8: Summary of the PCRE evaluation results.

Of three rule files shown in Table 8, there are total of 216 rules containing PCRE options, and 93 were successfully translated by TestingTool. This experiment demonstrated that TestingTool is able to reliably translate 43.1% of the Snort ruleset into bad packets that trigger the corresponding alert in Snort. This is due to the fact that the PCRE options in the other 123 rules contain PCRE Extended Patterns.

During the translation of the ruleset, if any of the PCRE Extended Pattern features are found in a rule, a warning will be given and TestingTool will skip it. In a regular PCRE expression parse tree, no matter which path is followed, it is ensured that a valid output exists. For example: `pcre:"(abc|def)";` if the abc path or def path is taken, a valid output is possible. However, for Extended Patterns, it may be possible that the given path does not have output when looking behind (`?<!pattern`). This makes the tree traversal more complex. For a positive look ahead of the problem, the object in the bracket may be a pattern itself (`?=a|b*`). There is no problem if the object is a simple string; however, catering for PCRE implies a greater complexity in TestingTool. Due to time constraints, TestingTool has not implemented Extended PCRE to address this current shortcoming.

Test case 5

Test case 5 has shown that TestingTool has failed this particular test. The attempt was to generate a different percentage of bad traffic for multiple protocols, by specifying a `-bad %` parameter after each protocol. This is one of the design goals of TestingTool: to be able to mix and match different parameters as stated in Chapter 4. Currently, the user is only allowed to specify one `-bad %` parameter on the command line. This is not a design flaw, but rather a bug that has been discovered late in the testing phase. Due to time constraints, this issue has not been resolved and the resolution to this problem is to allow the command line to handle multiple `-bad %` parameters.

5.3 Simulate Intrusion

Planned Test case

Test case 6: The user simulates 3 fragmentation and 2 portscan events to ensure the portscan and fragmentation simulations are functioning correctly.

Test case 7: The user generates 100 TCP segments with 20% of them to be bad along with 2 fragmentations and a portscan event. This ensures all three major functions are cooperating with each other accordingly.

Test case 8: The user inputs invalid characters to ensure that TestingTool validates the command line input.

Recorded Test Results

The following table is the recorded test results. The description and reasons for testing are mentioned beneath each of the tests recorded.

#	Command line Inputs ./Testing	Expected Output	Actual Output	Pass /Fail
Simulate Intrusion				
6	-Portscan 2 - Frag 3	2 attempts of portscan and fragmentation attack are simulated	Results were expected	P
Test that the fragmentation simulations are functioning correctly				
7	-TCP 100 -bad 20% - Portscan 1 -Frag 2	80 normal and 20 bad TCP packets are generated. 1 portscan and 2 fragmentation attempts simulated, causing 27 corresponding alerts to be outputted	Results were expected	P
Test that the three major functions work together correctly				
8	-Stream # \$ ^ - bad 0.1% - Frag 7a7	Invalid Input is rejected.	Results were expected	P
Test that invalid inputs are handled appropriately				

Observation

Test case 6

One of the requirements of TestingTool is to simulate attack instances (intrusion) in order to exercise different preprocessors in Snort. Test case 6 demonstrated that TestingTool is capable of simulating a portscan event with the port range 1-10,20,21,23,75:80,91-100,101,200 as specified in the configuration file. It can handle commas and dashes to specify the port range as initially designed. It was observed that this test case triggered a portscan alert from the `sfportscan` preprocessor in Snort. Moreover, as with the 3 fragmentation simulations that occurred at the same time, the corresponding fragmentation alerts arose as expected. Thus, TestingTool is capable of exercising the portscan and fragmentation preprocessor in Snort.

Test case 7

In order to determine Snort performance, it was necessary to analyse the behaviour of Snort using a variety of network traffic that have different characteristics. The cost of Snort analysing each packet varies, depending on the packet type and content of the payload. Test case 7 generated normal traffic containing ASCII payloads and bad traffic containing PCRE and binary payloads. Test case 7 utilised the three functions to produce a stream of traffic containing a combination of normal and bad traffic to exercise the signature matching engine and content matching engine. At the same time, two fragmentations and a portscan event were also simulated in order to further stress test the two individual preprocessors. It was observed that all corresponding alerts were raised and the cooperation of the three functions was as expected. The total size was 5,761,640 bytes, and it took 14.4 seconds for the test case to complete. Nemesis throughput is 3.2Mbit/sec in this test case.

Test case 8

For completeness, a test must be done with invalid inputs. Out of range values and special characters were entered as invalid inputs in Test case 8. It demonstrated that TestingTool has appropriate

error handling functions for dealing with special characters and invalid user inputs. The quality of TestingTool can be assured.

5.4 Comparison with specification

Three main requirements of TestingTool were summarized in Chapter 3. The extent to which each these requirements were achieved are described below.

First Requirement:

Generate Normal Traffic as specified in use case diagram figure 6.

Level of Achievement:

TestingTool allows the user to specify parameters such as Network protocols, number of packets, payload size and range of IP addresses which generate alert-free traffic matching those specifications accordingly. This requirement was achieved fully.

Second Requirement:

Generate Bad Traffic as specified in use case diagram figure 7.

Level of Achievement:

TestingTool allows the user to specify a percentage of traffic to be bad and is able to generate the correct amount of bad packets as specified.

TestingTool was able to parse Snort rules and translate the majority of rule options into the corresponding alert-trigger packets. The lack of full PCRE language support and multiple – bad percentage support means this requirement is only partially achieved.

Third Requirement:

Simulate Intrusion as specified in use case diagram figure 8.

Level of Achievement:

TestingTool allows the user to specify two types of intrusions and the number of attempts. TestingTool was able to generate the two types of intrusions, and both successfully triggered different pre-processor alerts.

5.5 Validation

As stated in Section 1.3, the aim of the project is to *develop a highly configurable and flexible testing tool that is able to generate different streams of network packets based on the rule sets and the parameters which the user provides*. TestingTool allow users to generate a mixture of good and bad traffic based on a rule file, a configuration file and command line arguments which the user provides. This satisfaction of specification requirements has been demonstrated above.

5.6 Comparisons with related product

Some comparisons can be made between TestingTool and some of the other NIDS testing application outlined in Chapter 2. Table 9 shows a comparison of other comparable applications with TestingTool.

	Emulate Full-session TCP-based attacks	Support Perl Compatible Regular expression	Support Mixed binary and ASCII content string	Emulate mix of normal and bad traffic	Exercise specific preprocessor
Whisker[15]	Supported	/	/	/	/
Mucus/Snot[16/5]	/	/	/	/	/
Agent[5]	Supported	/	/	/	Supported
Thor[14]	Supported	/	/	/	Supported
TestingTool	Supported	Supported	Supported	Supported	Supported

Table 9: Comparison of Previous Work with TestingTool

5.7 Original Contribution

The main contribution of this thesis was the development of a parser to handle the latest version of Snort signature. The more specific contributions are:

- The ability to translate Snort rule with a mixture of regular expression, binary and ACSII content string.
- A highly flexible and configurable design that allows the user to generate a combination of normal and bad traffic.

5.8 Self-Evaluation

The project plan in the progress report originally stated that, when the final product (TestingTool) is developed, performance testing of Snort will begin. A variety of test cases were going to be created for Snort performance evaluation. This part of the project plan was not met. The major reason for this was that the implementation of the PCRE language was more complex than expected. The documentation of the language is around 30 pages [24]. It was the implementation of the PCRE parser that accounted for the majority of the development time and caused a large underestimation.

The crafting of the Nemesis commands to trigger certain rule options also required more development time than anticipated. The tweaking of the Nemesis commands frequently took long hours and was not as obvious and straightforward as the examples given may seem. A typical example would be the `flow` option for both the client and server. It would initially seem simple; however, a simple modification for `flow` such as including an `established` option would require a completely different Nemesis command. It would require trial and error to test different flags, sources and destinations in order to trigger the `flow` option. There would be no consistent or predictable method of minimising the time required to determine the correct combination to trigger Snort rules containing the Flow option.

Apart from issues already mentioned which slowed development, most of the developments in the original stated plan were fulfilled. This was due to the fact that a lot of time and effort has been devoted into the making of the thesis.

5.9 Summary

This chapter presented an evaluation of the work completed for this thesis. The testing procedure for TestingTool was presented and some of the general observations made when testing the system were also discussed. An evaluation of TestingTool against the specifications derived in Chapter 3 was performed. This indicated that the requirements of the project were achieved to a reasonable level. Chapter 6 indicates some of the improvements that could be made to TestingTool in future versions.

6. Future Development

Chapters 3, 4 and 5 discussed the specification, implementation and evaluation of TestingTool produced in this study. In these chapters, comments were made regarding decisions that were necessary to ensure completion of the tool, but limited its flexibility or performance. In this chapter, these issues are analyzed with a look at what parts of the system could be improved for future versions. Consequently, possible improvements to the Generate Normal Traffic, Generate Bad Traffic and Simulation Intrusion functions are all discussed as possible new features.

6.1 Generate Normal Traffic

The first improvement would be to increase Nemesis performance. This would allow Nemesis to produce traffic with higher throughput which in turn would be able to stress test Snort's performance and capacity. As mentioned in the chapter 5 Evaluation, Nemesis spawn a new process for each packet. The throughput of Nemesis can be improved by modifying the Nemesis source code to eliminate the start-up and tear-down overhead of spawning a new process.

A more significant improvement would be to have full support for all Snort rule options, including Extended PCRE. The existing architecture can support such changes. The current options support uses an options list to add or remove new options to the rules, which allows a potentially unlimited number of options to be added without changing the structure of the rule itself.

Furthermore, the processing of each option could be improved greatly by implementing a table of options along with the function addresses for function handlers. Such implementation would allow greater flexibility in the set of options handled and make code development easier to understand and support.

6.2 Generate Bad Traffic

The scope of the project, together with the limited time available to complete it, has meant that many design and implementation issues for traffic generation have not been studied in significant detail. The generation of bad traffic does not need to be limited to the parsing of snort rules and creating packets that conform to the rules. Instead of translating a Snort rule, a completely different approach for generating more realistic network traffic would be to utilise a variety of network traces. Network traces contain real traffic that is captured on a set of peering points from a wide area network. Article Evangelos and Markatos [11] suggested a number of network traces from different environments, such as `UCNET` and `FORTH.web`. `UCNET` are traces from the ATM link between the University of Crete and GRNET. `FORTH.web` is a trace containing a number of concurrent Web browsing sessions captured at FORTH. Roughly 42% of packets in the `FORTH.web` trace trigger a set of 956 rules in Snort. A set of traces will be selected for use as a benchmark when performance testing a NIDS. This approach is better in the way that the traffic generated is more realistic by containing various payload sizes and content. It can provide an accurate estimate of NIDS capacity under different scenarios. The TestingTool architecture supports such changes since there is a clean interface between the traffic generator and the parser. Any future changes with respect to the traffic generator can be made in a straightforward manner. If TestingTool was extended to be able to utilise traces and parse Snort rules for bad traffic generation, then the developer would be able to try both implementations and compare the benefits of each alternative.

6.3 Simulate Intrusion

The intrusion simulation function could be extended by designing more types of intrusion to exercise more preprocessors in Snort. An additional feature could be to simulate the Distributed Denial of Service (DDOS) Attack for exercising the Stream4 preprocessor. By also using the obfuscation technique mentioned in Chapter 2, it is possible to exercise another preprocessor called the HTTP decoder.

7. Conclusion

This thesis has presented the details of the development and evaluation of `TestingTool`. It is a highly configurable and flexible traffic generation tool for evaluating a Network Intrusion Detection System.

The `TestingTool` application is the main outcome for this study. It is able to generate different streams of network packets based on a set of parameters the user provides. There are three core functionalities in `TestingTool`. First, `TestingTool` is capable of generating normal traffic. It creates alert-free traffic by crafting simple `Nemesis` commands for various protocols. Second, `TestingTool` is capable of generating bad traffic. It creates alert-triggering traffic by translating `Snort` rules into series of packets that conform to the `Snort` rules. Third, `TestingTool` is capable of simulating actual intrusions to test the different preprocessors in `Snort`.

A formal software evaluation has shown that the majority of the requirements of `TestingTool` were met. It can reliably produce network traffic according to the various parameters the user has entered into the tool.

There are elements of the application that can be improved upon. It is possible to use network traces to generate bad traffic instead of transmitting `Snort` rules. `Nemesis` throughput can be improved by reducing the overhead of spawning a new process for every packet generated. Full option support including Extended PCRE would enable `TestingTool` to handle more rules. It is also desirable in the future for `TestingTool` to simulate more types of intrusions such as DDOS attacks to test other preprocessors within `Snort`.

This thesis describes the processes followed to produce a working `TestingTool` application. The motivation of the study was outlined in the first chapter. The purposes of the thesis were outlined in this chapter with the goals decided upon. Afterwards, a summary of the previous work done was given. This provided some basic theories

that were drawn upon during the design process. The project was then formally scoped and the software requirements were presented as use cases. The process of design and implementation for TestingTool was then given, including the design breakdown that was used for the three main functions, as well as reasons behind the major design decisions that were made. The results of testing the system were then presented and the design evaluated against the given specifications as well as compared to other related projects. Future directions for TestingTool were then outlined.

References

- [1] M. Ranum, "Experience Benchmarking Intrusion Detection Systems," Dec 2001. <http://www.itsecurity.com/archive/papers/nfr2.htm>, last accessed 07/09/06.
- [2] Next Generation Software Security Ltd. NGSS IDS Evaluation (4th Edition), 2004. <http://www.nss.co.uk/ids/edition4/index.htm>, last accessed 07/09/06.
- [3] J. B. Raven Alder, Adam Coxtater, James C. Foster, Toby Kohlenberg, & Michael Rash, *Snort 2.1 Intrusion Detection*, Second ed. Massachusetts: Syngress, 2004.
- [4] T. Thomas, & H. Ptacekm, "*Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*," Technical Report T2R-0Y6, Secure Network, Inc., Calgary, Alberta, Canada, 1998.
- [5] S. J. S. Rubin, & B.P. Miller, "Automatic generation and analysis of NIDS attacks," in *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, AZ, USA, IEEE Computer Society, Dec. 2004.
- [6] D. Prawitz. *Natural deduction, a proof-theoretical study*. Almqvist & Wiksell, Stockholm, 1965.
- [7] W. R. Giovanni Vigna, & Davide Balzarotti, "Testing Network-based Intrusion Detection Signatures Using Mutant Exploits," in *Proceedings of the 11th ACM conference on Computer and communications security*, Washington DC, USA October 25-29, 2004.
- [8] "Stick can be used as a denial of service tool against intrusion detection systems", 2001. <http://xforce.iss.net/xforce/xfdb/6552>, last accessed 07/09/06
- [9] S. Aubert., "Idswakeup", 2000. www.hsc.fr/ressources/outils/idswakeup/index.html.en, last accessed 07/09/06

- [10] K. G. A. Spyros Antonatos, & Evangelos P. Markatos, "Generating Realistic Workloads for Network Intrusion Detection Systems," in *Proceedings of the 4th international workshop on Software and performance WOSP '04*, Redwood Shores, California January 14-16 , 2004.
- [11] Joel Sommers & Paul Barford. "Self-configuring network traffic generation," In *IMC'04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 68-81, New York, NY, USA, October 2004. ACM Press.
- [12] QoD, "A look into IDS/Snort," 2004,
<http://www.antonline.com/showthread.php?s=&threadid=253920>, last accessed 07/09/06
- [13] M. Holstein, "Intrusion Detection FAQ: How does Fragroute evade NIDS detection?," 2002.
<http://www.sans.org/resources/idfaq/fragroute.php?portal=75bacb4b3fbf07f754732f60088bf5d6>, last accessed 07/09/06
- [14] R. Marty, *THOR : A Tool to test Intrusion Detection System by Variations of Attacks*, Diploma Thesis: ETH Swiss Federal Institute of Technology Zurich, 1 March 2002.
- [15] Rain Forest Puppy. "A look at whisker's anti-IDS tactics – just how bad can we ruin a good thing?," December 1999.
www.wiretrip.net/rfp/txt/whiskerids.html, last accessed 07/09/06
- [16] D. Mutz, G. Vigna, & R. Kemmerer. "An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems." In *Proceedings of Annual Computer Security Applications Conference*, Las Vegas, NV, December, 2003.
- [17] S. Siddharth, "Evading NIDS, revisited," 2005.
<http://www.securityfocus.com/infocus/1852>, last accessed 07/09/06
- [18] M. Handley & V. Paxson. "Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics." In *Proceedings of USENIX Security Symposium*, Washington, DC, August 2001.

- [19] U. Shankar & V. Paxson. "Active mapping: Resisting NIDS evasion without altering traffic." In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003
- [20] D. Mutz, G. Vigna, & R. Kemmerer, "An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems," In *proceedings of ACSAC '03*, Las Vegas, NV, December 2003.
- [21] Renaud Deraison, Jay Beale, HD Moore, Noam Rathaus, et al, *Nessus Networking Auditing*, Massachusetts: Syngress, 2004.
- [22] The Snort Project, "Snort Manual 2.6.1," 2004.
www.snort.org/docs/snort_manual/2.6.1/snort_manual.pdf, last accessed 07/04/07
- [23] J.Nathan, "Nemesis documentation," 2002.
<http://nemesis.sourceforge.net/docs.html>, last accessed 29/05/07
- [24] P. Hazel, "Perl version 5.8.8 documentation," 2003.
<http://perldoc.perl.org/perlre.html>, last accessed 4/04/07
- [25] Chris McNab, *Network Security Assessment, Know Your Network*. O'Reilly, March 2004.

Appendix A: Grammar rule for Snort Signature

Parser uses the following grammar rule for parsing Snort rules:

```
rules_file
    : list_of_entries
    ;

list_of_entries
    : var_entry list_of_entries
    | rule_entry list_of_entries
    ;

var_entry
    : VAR IDENTIFIER string_of_characters
    ;

rule_entry
    : rule_header options
    | rule_header
    ;

rule_header
    : action proto address direction address
    ;

options
    : '(' option_list ')'
    ;

proto
    : TCP
    | UDP
    | ICMP
    .....
    | NUMBER
```

```

;

action
  : LOG
  | DENY
  | ALERT
  ;

address
  :ip_part port_part
  |ip_part
  ;

ip_part
  :ip '/' MASK
  |ip
  ;

port_part
  : port
  | NUMBER ':' NUMBER
  ;

ip
  : IP
  | any
  ;

port
  : NUMBER
  | any
  ;

option_list:
  single_option ';'
  | single_option ';' option_list
  ;

```

```
single_option:
    IDENTIFIER
    | IDENTIFIER ':' VALUE
    | IDENTIFIER ':' sign VALUE
    ;
```

```
sign:
    '<'
    | '>'
    | '='
    | '<='
    | '>='
    ;
```

Appendix B: Configuration File Used for Testing

```
# Set up the home network addresses.
var HOME_NET 10.1.1.1

# Set up the external network addresses.
var EXTERNAL_NET 192.168.2.0/24

# Configure server lists. This allows TestingTool to parse Snort
rules
# that contain these variables.

# List of DNS servers on your network
var DNS_SERVERS $HOME_NET

# List of SMTP servers on your network
var SMTP_SERVERS $HOME_NET

# List of web servers on your network
var HTTP_SERVERS $HOME_NET

# List of sql servers on your network
var SQL_SERVERS $HOME_NET

# List of telnet servers on your network
var TELNET_SERVERS $HOME_NET

# List of snmp servers on your network
var SNMP_SERVERS $HOME_NET

# Configure service ports.
#
# var HTTP_PORTS 8081
#
# Port lists must either be continuous [eg 80:8080], or a single port
[eg 80].
```

```
# We will adding support for a real list of ports in the future.

# Ports that web servers run on
#
# Please note: [80,8080] does not work.
# If you wish to define multiple HTTP ports, use the following
convention
# when customizing rule set.
#
var HTTP_PORTS 80

# The starting port number for tcp and udp normal traffic
var PORT 12345

# The rule file to be used to generate bad traffic
var BAD local.rules

# No payload file will be used when "disable" is being specified.
# payload_file is 1.2kB in size
var payload payload_file

var portscan 1-10,20,21,23,75:80,91-100,101,200
#var portscan vquick
#var portscan quick
#var portscan all

var fragmentation frag.rules

var output output
```

Appendix C: Rule File (local.rules)

Used for Testing

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 465 ISS
Pinger"; itype:8; content:"ISSPNGRQ"; depth:32;
reference:arachnids,158; classtype:attempted-recon; sid:1001; rev:4;)
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 471 icmpenum
v1.1.1"; dsize:0; icmp_id:666 ; icmp_seq:0; id:666; itype:8;
reference:arachnids,450; classtype:attempted-recon; sid:1002; rev:4;)
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 474
superscan echo"; dsize:8; itype:8; content:"|00 00 00 00 00 00 00
00|"; classtype:attempted-recon; sid:1003; rev:5;)
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 480 PING
speedera"; itype:8; content:"89|3A 3B|<=>?"; depth:100;
classtype:misc-activity; sid:1004; rev:6;)
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 481
TJPingPro1.1Build 2 Windows"; itype:8; content:"TJPingPro by Jim";
depth:32; reference:arachnids,167; classtype:misc-activity; sid:1005;
rev:6;)

alert udp any any -> any 69 (msg:"TFTP 1289 GET Admin.dll";
content:"|00 01|"; depth:2; content:"admin.dll"; offset:2; nocase;
reference:url,www.cert.org/advisories/CA-2001-26.html;
classtype:successful-admin; sid:2001; rev:4;)
alert udp $EXTERNAL_NET any -> $HOME_NET 69 (msg:"TFTP 520 root
directory"; content:"|00 01|/"; depth:3; reference:arachnids,138;
reference:cve,1999-0183; classtype:bad-unknown; sid:2002; rev:5;)
alert udp $EXTERNAL_NET any -> $HOME_NET 69 (msg:"TFTP 518 Put";
content:"|00 02|"; depth:2; reference:arachnids,148;
reference:cve,1999-0183; classtype:bad-unknown; sid:2003; rev:6;)
alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS 253 SPOOF query
response PTR with TTL of 1 min. and no authority"; content:"|85 80 00
01 00 01 00 00 00 00|"; content:"|C0 0C 00 0C 00 01 00 00 00|<|00
0F|"; classtype:bad-unknown; sid:2004; rev:4;)
alert udp $EXTERNAL_NET 53 -> $HOME_NET any (msg:"DNS 254 SPOOF query
response with TTL of 1 min. and no authority"; content:"|81 80 00 01
```

```
00 01 00 00 00 00|"; content:"|C0 0C 00 01 00 01 00 00 00|<|00 04|";
classtype:bad-unknown; sid:2005; rev:4;)
```

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:1; id:5;
content:"3001"; msg:"3001"; sid:3001;)
```

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:2; id:6;
content:"3002"; msg:"3002"; sid:3002;)
```

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:3; id:7;
content:"3003"; msg:"3003"; sid:3003;)
```

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:4; id:8;
content:"3004"; msg:"3004"; sid:3004;)
```

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:5; id:9;
content:"3005"; msg:"3005"; sid:3005;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (ttl:=3; flags: SF,12;
seq:1234; ack:5687; msg:"4001"; sid:4001;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any ( ttl:<1; flags: SF,12;
window:!15; seq:0; msg:4002; sid:4002;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any ( ttl:>=255; flags:
SF,12; window:23; msg:4003; sid:4003;)
```

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (ttl:2; flow:
from_client,established; tos:5; window: 2345; msg:4004; sid:4004;)
```

```
alert tcp $HOME_NET any -> $EXTERNAL_NET any (ttl:3; flow:
established,to_client; msg:"4005"; sid:4005;)
```

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 482 PING
WhatsupGold Windows"; itype:8; content:"WhatsUp - A Netw"; depth:32;
reference:arachnids,168; classtype:misc-activity; sid:1006; rev:6;)
```

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 483 PING
CyberKit 2.2 Windows"; itype:8; content:"|AA AA AA AA AA AA AA AA AA
AA AA AA AA AA AA AA|"; depth:32; reference:arachnids,154;
classtype:misc-activity; sid:1007; rev:6;)
```

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 484 PING
Sniffer Pro/NetXRay network scan"; itype:8; content:"Cinco Network,
Inc."; depth:32; classtype:misc-activity; sid:1008; rev:5;)
```

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 1813 digital
island bandwidth query"; content:"mailto|3A|ops@digisle.com";
depth:22; classtype:misc-activity; sid:1009; rev:6;)
```

```

alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP 1010";
content:"echo reply echo reply"; depth:22; sid:1010; rev:6;)

alert udp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS 314 EXPLOIT
named tsig overflow attempt"; content:"|80 00 07 00 00 00 00 00
01|?|00 01 02|"; reference:bugtraq,2303; reference:cve,2001-0010;
classtype:attempted-admin; sid:2006; rev:9;)
alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC 1923 portmap
proxy attempt UDP"; content:"|00 01 86 A0|"; content:"|00 00 00 05|";
distance:4; content:"|00 00 00 00|"; sid:2007;)
alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC 1280 portmap
listing UDP 111"; content:"|00 01 86 A0|"; content:"|00 00 00 04|";
distance:4; content:"|00 00 00 00|"; sid:2008;)
alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC 1950 portmap
SET attempt UDP 111"; content:"|00 01 86 A0|"; content:"|00 00 00
01|"; distance:4; content:"|00 00 00 00|"; sid:2009;)
alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC 2015 portmap
UNSET attempt UDP 111"; content:"|00 01 86 A0|"; content:"|00 00 00
02|"; within:4; content:"|00 00 00 00|"; sid:2010;)

alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:6; id:10;
content:"3006"; msg:"3006"; sid:3006;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:7; id:11;
content:"3007"; msg:"3007"; sid:3007;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:8; id:12;
content:"3008"; msg:"3008"; sid:3008;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:9; id:13;
content:"3009"; msg:"3009"; sid:3009;)
alert ip $EXTERNAL_NET any -> $HOME_NET any (tos:10; id:14;
content:"3010"; msg:"3010"; sid:3010;)

alert tcp $HOME_NET any -> $EXTERNAL_NET any (ttl:2; flow:
established,from_server; window:5555; msg:"4006"; sid:4006;)
alert tcp $HOME_NET any -> $EXTERNAL_NET any (ttl:4; flow:
established,to_server; window:123; msg:"4007"; sid:4007;)
#gd threshold has been removed
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"SPYWARE-
PUT 5831 Hijacker comet systems runtime detection - update requests";

```

```

flow:to_server,established; uricontent:"v="; nocase; uricontent:"t=";
nocase; uricontent:"c="; nocase; content:"Host|3A|"; nocase;
content:"update.cc.cometsystems.com"; distance:0; nocase;
pcre:"/\x2F[^\r\n]*\.(dat)|(xml)\?[^\r\n]*v=[^\r\n]*t=[^\r\n]*c=/Ui";
pcre:"/^Host\x3A[^\r\n]*update\.cc\.cometsystems\.com/smi";
reference:url,www.spywareguide.com/product_show.php?id=428;
reference:url,www3.ca.com/securityadvisor/pest/pest.aspx?id=453088065;
classtype:misc-activity; sid:4008; rev:1;)

#gd
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-
CGI 1455 calendar.pl access"; flow:to_server,established;
uricontent:"calendar"; nocase; pcre:"/calendar(|[_]admin)\.pl/Ui";
reference:bugtraq,1215; reference:cve,2000-0432; classtype:attempted-
recon; sid:4009; rev:7;)

#gd also trigger 1:1112:6
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-
CGI 3674 db4web_c directory traversal attempt";
flow:to_server,established; uricontent:"/db4web_c"; nocase;
pcre:"/db4web_c(\.exe)?\/*(\.\.[\\|\/]| [a-z]\:)/smiU";
reference:bugtraq,5723; reference:cve,2002-1483;
reference:nessus,11182; classtype:web-application-attack; sid:4010;
rev:1;)

```

Appendix D: Supported PCRE

TestingTool supports regular expression:

*	Match 0 or more times	
+	Match 1 or more times	
?	Match 1 or 0 times	
{n}	Match exactly n times	
{n,}	Match at least n times	
{n,m}	Match at least n but not more than m times	
*?	Match 0 or more times	
+?	Match 1 or more times	
??	Match 0 or 1 time	
{n}?	Match exactly n times	
{n,}?	Match at least n times	
{n,m}?	Match at least n but not more than m times	
\t	tab	(HT, TAB)
\n	newline	(LF, NL)
\r	return	(CR)
\f	form feed	(FF)
\a	alarm (bell)	(BEL)
\e	escape (think troff)	(ESC)
\033	octal char (think of a PDP-11)	
\x1B	hex char	
\x{263a}	wide hex char	
\c[control char	
\N{name}	named char	
\l	lowercase next char (think vi)	
\u	uppercase next char (think vi)	
\L	lowercase till \E (think vi)	
\U	uppercase till \E (think vi)	
\E	end case modification (think vi)	
\Q	quote (disable) pattern metacharacters till \E	
\w	Match a "word" character (alphanumeric plus "_")	
\W	Match a non-"word" character	
\s	Match a whitespace character	
\S	Match a non-whitespace character	

`\d` Match a digit character
`\D` Match a non-digit character
`\pP` Match P, named property. Use `\p{Prop}` for longer names.
`\PP` Match non-P
`\X` Match eXtended Unicode "combining character sequence",
 equivalent to `(?:\PM\pM*)`
`\C` Match a single C char (octet) even under Unicode.
 NOTE: breaks up characters into their UTF-8 bytes,
 so you may end up with malformed pieces of UTF-8.
 Unsupported in lookbehind.

Classes supported are:

alpha		
alnum		
ascii		
blank		[1]
cntrl		
digit	<code>\d</code>	
graph		
lower		
print		
punct		
space	<code>\s</code>	[2]
upper		
word	<code>\w</code>	[3]
xdigit		

The following Extended PCRE patterns are not supported.

- `(?#text)`
- `(?imsx-imsx)`
- `(?:pattern)`
- `(?imsx-imsx:pattern)`
- `(?=pattern)`
- `(?!pattern)`
- `(?<=pattern)`
- `(?<!pattern)`
- `(?{ code })`
- `(??{ code })`

- `(?>pattern)`
- `(?(condition)yes-pattern|no-pattern)`
- `(?(condition)yes-pattern)`