MASTER

Towards better training sets for intrusion detection systems

van der Wiel, J.H.

Award date:
2011

Towards better training sets for Intrusion Detection Systems

J.H. v.d. Wiel

June 2011

Technische Universiteit Eindhoven

Master Thesis

Presented By

J.H. v.d. Wiel

Towards better training sets for Intrusion Detection Systems

prof. dr. A. E. Brouwer	TU/e
prof. dr. S. Etalle	TU/e, Universiteit Twente
prof. dr. S. Zanero	Politecnico di Milano

Contents

List of Figures

List of Tables

Abstract

The most common way to determine the quality of a system is to test it. This also holds for Intrusion Detection Systems (IDSs). Throughout the years various research has been conducted on testing IDSs. However the data used for testing an IDS is relatively old, and the focus has been for the last couple of years on IDS evasion techniques. Techniques how to create benign data are often fairly limited in terms of functionality and quality. Currently the most advanced way is the usage of `wget`, which has quite some drawbacks (small think time window, easily detectable by verifying the version, number of requests made) when it is used as the sole source for benign traffic.

In this thesis we developed a tool to create benign artificial traffic which functions better than `wget`. Therefore we developed a tool that consists of 2 parts: a protocol analysis part and a traffic generation part. Although the techniques can possibly be applied to a variety of protocols, we focused on HTTP since this is the most used protocol for which the most vulnerabilities are found.

The protocol analysis part analyzes a few sample sessions. It obtains information on how to (partly) parse the protocol as well as information on how to construct requests.

The traffic generation part uses the information from the protocol analysis part. It combines the data from several sessions into a semi state machine. This semi state machine is a program that pretends to be a client and interacts with a server by sending requests and receiving responses.

Within the defined test cases our tool is proven to work. The used IDSs do not raise an alarm when our tool is used. A script written to detect `wget` however does raise an alarm when wget is used. No alarm is raised when the data from our tool is used.

Acknowledgements

I thank my supervisors prof. dr. A. E. Brouwer, prof. dr. S. Etalle, and dr. S. Zanero for their valuable support. Their comments were always very constructful. Also I would have been stuck on many different subjects if it wasn't for them.

Besides my supervisors 2 other persons: dr. D. Bolzoni and dr F. Maggi helped me.

I also thank my friends who provided me with sample data to play with: C. Aleva, M. Aleva, V. Y. Nguyen, F. v. Vliet.

Last but not least I thank my girlfriend for supporting me.

Chapter 1

Introduction

This chapter gives an introduction into this thesis. First the motivation for writing this thesis is stated in Section 1.1, which stresses the importance of computer security in general. The next section, Section 1.2, provides the problem statement, followed by our contribution in Section 1.3. Finally, Section 1.4 gives an outline for the rest of this thesis.

1.1 Motivation

Information in today's society is of great importance, which is shown by the fact that today's society is an information society where information is the most important business asset [7]. Of course sensitive information has to be protected. Failing to do so can result in a loss of money [8] or can have geopoliticial consequences [9].

One way to detect an intrusion is by using Intrusion Detection Systems (IDSs) of which the definition as well as more information can be found in Section 2.1.

In general testing of critical systems is important since it helps with finding different types of flaws such as security and design flaws. Besides that, it also helps with measuring the performance of the system. Testing, and deriving conclusions from it, stands or falls with the quality of the input. This input, from now on referred to as the input data, can differ depending on the test aim. For example, for finding security flaws different input data can be required than when the correctness of the system is determined, even though they are not mutually exclusive.

In the case of IDS testing other problems are added. For example, there is the bandwith problem: An anomaly based IDS (see Section 2.1.3), can be trained with input data which has limited bandwith usage. However, when the IDS is used in a live environment, then it is possible that the bandwith of the analyzed traffic is much larger than that of the input data. This can lead to FP's, etc (see Section 2.1.1) [10].

Another more specific problem is the analysis of data for anomalies. This is usually done by other IDSs. If the other IDSs do not see an attack, then the IDS that is trained with this input data will detect it neither, or when detected it is flagged as a true negative[10].

A more fundamental problem is privacy issues. Real world data contains information specified by users. This information can be anonymized, resulting in more problems. For example, when all the email addresses in a dataset are replaced with the same email address, then this leads to unrealistic traffic[11].

1.2 Problem statement

In this thesis we focus on training sets for IDSs. Part of chapter 2 contains an analysis of related work considering training sets. From that chapter the following points are of importance for the rest of this section:

- The best way, currently available, to create host specific benign artificial traffic for IDSs that analyze the application layer, is to use `wget -m` (a program that mirrors, with this option, a web/ftp site). An IDS can spot this quite easily by analyzing for example the think time and/or the client version.
- There is relatively a lot of focus on IDS evasion, for which quite some tools are available with a fair amount of documentation. This in contrast to tools that create benign artificial traffic, for which almost no tools are freely available and for which the documentation is often quite minimal.

In short: The best way to create host specific benign artificial traffic is currently achieved in a rather simplistic way. This in comparison to IDS evasion techniques, which are quite advanced and well documented. In this thesis we try to overcome these shortcomings.

1.3 Our contribution

We aim to develop a tool that can be used to create host specific benign artificial data and overcomes the aforementioned problems. Our tool will consist of two parts: a plain text protocol analysis part and a traffic generation part which uses the information from the protocol analysis part. The protocol analysis part is created in such a way that it can possibly be applied to a variety of plain text protocols, since many plain text protocols share the same characteristics[1].

Analyzing an application layer protocol and creating traffic for that layer has some advantages. It comes for example, the closest to emulating a real user, since a typical user has no direct influence on the underlying protocols. The behavior of the underlaying thus do no require an implementation. However these layers can create malicious or remarkable data. An example is broken packets, as can be found at [12]. They have not been created directly by a user, but more likely by faulty routers, broken IP stack implementations, etc.

For the analysis part, a few sample sessions are enough to obtain the correct and useful information, making analyzing large quantities of data superfluous. From these sample session statistics are obtained, as well as information on how to (partly) parse the protocol and how to find important information from responses. For example, when a client sends a request to a server, the client receives a response. The next request can depend on a response received earlier,

[1]In this thesis we only focus on HTTP however

which in turn depends on a previous request. Now, not the whole request is saved, but information on how to construct the request is. This means that part of the information is which request has to be sent first, in order to obtain the data to construct this new request, and also where this information can be found in the response, etc.

The big advantage of this approach is that when the information in the responses change, a valid and logical (since the content of this request depends on content received in this session and not on the content of a previous session) request can still be created.

At this point it is possible to “replay” traffic at a later point in time, since all the information on how to construct the requests is available. Creating large quantities of unique sessions is however not possible. To make this possible different sessions are mixed (semi randomly), resulting in a semi state machine. This semi state machine is a small program, able to interact with a server and pretending to be a real client. It is the traffic generation part of our tool. Upon starting the program a session is created with the server and benign traffic is created by sending various request to the server and parsing useful information. Each semi state machine depends on the randomization that is used, meaning if the randomization is different, a different semi state machine is created.

The advantage of creating a program lies in the fact that it is portable. It can be easily distributed to many machines and thus coming closer to real user simulation (in comparison to a testbed for example).

In this thesis we focus, as mentioned before, on the HTTP protocol. The semi state machine is also designed for this protocol (a request-response protocol). For other protocols such as an Instant Message (IM) protocol, another type of semi state machine is required.

1.4 Organisation of this thesis

Chapter 2 gives an introduction into the subject of IDSs (Section 2.1), followed by Section 2.2, which discusses various approaches to test IDSs. The last section in that chapter is Section 2.3, which discusses various types of input data or ways to create input data, for IDS evaluation. Chapter 3 discusses algorithms and techniques to analyze plaintext protocols of which no information is known and determines which information is required to create the semi state machine. The next chapter, Chapter 4 discusses some techniques for mixing various sample sessions into a semi state machine and gives more information on the semi state machine itself. Next, Chapter 5 describes several test cases for testing the tool and also gives the outcome of these tests. Finally, Chapter 6 states a conclusion and discusses future work.

While writing this thesis we also developed a statistical distribution. At a later point in time we decided not to use this. It can be found in Appendix C

Chapter 2

Related work

This chapter is divided into 4 parts. The first part, Section 2.1, gives an introduction into IDSs after which Section 2.2 discusses various topics on IDS testing. Next, Section 2.3 discusses several types of input data aimed at helping IDS evaluation. Finally, Section 2.4 states a conclusion.

2.1 Introduction

This section consists of 2 parts. First a general introduction is given in Section 2.1.1, after which the rest of this section is used to discuss various types of IDSs.

2.1.1 Intrusion Detection Systems

Intrusion Detection Systems were first introduced in 1980 by J.P. Anderson [13]. The National Institute of Standards and Technology (NIST) defines an IDS as follows: "IDSs are software or hardware systems that automate the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems"[14]. Intrusion detection is defined by the NIST as: "The process of monitoring the events occurring in a computer system or network and analyzing them for signs of intrusions, defined as attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer or network"[14].

The latter definition introduced the terms: confidentiality, integrity and availability which are defined as [15]:

Confidentiality requires information to be kept private.

Integrity is the trustworthiness and correctness of data.

Availability is the capability to use information and resources.

A common feature of IDSs is to raise alarms upon detection of attacks, after which appropriate actions can be taken. Other functionality can include logging attacks, and recognize violations of an organization's security policy [16].

The following quantities are the possible results of intrusion detection:

True Positive An alarm is raised for real intrusion attempts.

True Negative No alarm is raised when no intrusion is attempted.

False Positive An alarm is raised when no intrusion is attempted.

False Negative No alarm is raised for real intrusion attempts.

A general drawback of IDSs is that detecting intrusions in all cases is impossible. This was proven by Fred Cohen, as early as 1984 [17]. Detecting an anomaly can be as hard as the halting problem. As a consequence, the amount of resources to catch anomalies grows with the amount of usage of the system. Another drawback is that intrusions are detected by definition and not prevented. The adversary is able to obtain the protected data. Therefore Intrusion Prevention Systems (IPSs) are made which can be seen as extensions of IDSs. However this brings along other problems such as False Positives (FPs), which can be inconvenient for the end user.

A typical IDS consists of the following components [16]:

Sensors/Agents monitor and analyze activity. Typically the term sensor is used for Network based IDSs (see Section 2.1.4), while the term is agent is typically used for host based IDS (see Section 2.1.4).

Management server receives information from the sensor/agents and manages this information. Management servers can perform tasks that agents/sensors cannot, such as correlate information from various agents/sensors. Although a management server is not a required component for an IDS, it is still used in large IDS setups.

Database server functions as a repository for information recorded by sensors/agents and or management servers.

Consoles provide the interface for IDS users and administrators.

2.1.2 Misuse based IDSs

A misuse based IDS analyzes data streams for known patterns (signatures). These signatures describe known attacks. A misuse based IDS is therefore also known as a signature based IDS. Typically each signature describes a specific attack. More advanced techniques are available to detect groups of attacks[14]. It is also the most simple detection method since data or behaviour is compared against string patterns. Typical signature based IDSs also have little understanding of protocols. Therefore attacks that comprise multiple events are not always detected [16].

Advantages of misuse based IDSs include:

- Almost no false alarms are raised.
- A specific attack or tool is quickly and reliably recognized, which can help administrators take specific countermeasures.
- Easy to design.

Disadvantages are:

- Continuous updating of signatures is required.
- Cannot detect new attacks.

2.1.3 Anomaly based IDSs

An anomaly based IDS identifies abnormal behavior (anomalies). Traffic is compared using various (statistical) methods with profiles. These profiles are created during the training phase. During this phase all the traffic that is analyzed is legitimate [14][18]. However, due to the nature of Internet traffic (it is dynamic), new profiles need to be created periodically. A solution would be to use dynamic profiles. Unfortunately these are not really safe due to their design. An example is an adversary who slowly increases her malicious activity until it is at the required level. Another problem is that during the initial training phase only benign traffic is allowed which, depending on the used input data, can require a fair amount of auditing.

Advantages of anomaly based systems are:

- It can detect unknown attacks.
- Does not require continuous updating.
- Alerts can be used to create signatures for signature based IDSs.

Disadvantages are:

- Many false alarms are raised.
- Difficult to design.
- The training phase takes a long time.

2.1.4 Other types of IDSs

Section 2.1.2 and 2.1.3 stated the two main categories in which IDSs can be divided. However IDSs can also be categorized based on the analyzed data source and on the design of the IDS itself [14]:

Network Based A Network Based IDS (NIDS) captures and analyzes packets at the network segment that are analyzed for multiple hosts. Typically however, most focus lies on the application layer.

Host Based A Host Based IDS collects data from within a single computer system. The data contains for example system calls, system logs, etc.

Centralized Based In a Centralized Based approach, all analyzing is controlled from a central location.

Distributed Based In a Distributed Based approach there are agents that report to the point of analysis, at which a response decision is made.

2.2 IDS testing approaches

This section is divided into 3 parts. First, in Section 2.2.1, an introduction is given after which Section 2.2.2 discusses what requirements the input data needs to fulfill. Finally, Section 2.2.3 discusses how the outcomes of tests can be interpreted.

2.2.1 Introduction

Testing is of vital importance for systems that are not proven to work. With testing specific aspects of a system can be tested. For example, in the case of an IDS, the throughput and the security of the IDS can be tested. Every test objective can require different input data. For example, for testing the security of an IDS, a fuzzer can be used while for testing its throughput other data can be used.

The test result depends on the input data. This implies that when the system is tested for security bugs and the input data does not contain data to trigger a flaw present in the system, the test result yields secure, while in fact the system is not secure. Thus the test result, and the conclusion derived from it, stands or falls with the input data.

2.2.2 Input data requirements

As mentioned in the previous section, each test objective requires input data. This implies that in the worst case scenario, each test objective requires different input data. When this scenario is taken even further it means that each type of IDS, for each test scenario, needs its own input data. For an anomaly NIDS this implies that every NIDS, for each host, requires its own host specific input data.

For a scientific evaluation of a product it is important that the test is repeatable, based on rational, open, disclosed, unbiased standards and that it is scientifically sound [19]. Especially the first requirement puts some constrains on the input data. It has to be recreatable, or static, or saved (during the first test).

Section 2.2.1 stated that the input data has to contain data aimed at testing a certain aspect of the IDS. A result of this requirement is that in some cases the input data needs to be well analyzed. For example, in the case of an anomaly NIDS, attacks and benign traffic need to be well determined in order to derive a proper conclusion from testing the IDS.

2.2.3 Testing

The results of testing an IDS also depends on the rule set. For example, a NIDS placed in front of a webserver running Apache can be configured to catch successful exploits of an IIS webserver. However in reality this will never happen because the Apache server is not vulnerable to IIS exploits. If during the testing phase a data set is used which contains a successful IIS exploitation, then the test results are not correct, since in reality this can never happen.

Once the rules have been defined, the results have to be interpreted. One option is to count FPs, FNs, etc. This approach is particularly useful for anomaly IDSs, while it is not useful for signature based IDSs. This because a signature based IDS only counts positives and negatives.

Based on different tests and using FPs, FN's, etc, ROC-curves (plots the TP rate v.s. the TN rate) can be made. These are useful for anomaly IDS testing, since they help with finding the optimal parameters.

2.3 IDS test data

This section discusses several types of input data aimed at helping IDS evaluation. First background traffic generation is discussed in Section 2.3.1 after which Section 2.3.2 discusses several data sets. Next, Section 2.3.3 discusses frameworks, followed by Section 2.3.4 which discusses attack mutation tools. Although the related work is categorized in the aforementioned categories, it does not mean that they are mutually exclusive.

2.3.1 Background traffic generation

Originally traffic generation was used to test network devices. Usually pseudo random data was used as the payload. However, IDSs today analyze the payload of the protocol layer. These types of traffic generators are therefore not applicable for IDS testing and omitted in this chapter. As a solution smart traffic generators were introduced that sometimes (partly) implement various protocols.

The rest of this section describes and comments on various kinds of background traffic generators.

Harpoon

Harpoon is a tool developed by Sommers et al in 2004. It generates representative traffic at the Internet layer level. Both TCP and UDP packet flows can be generated of which some characteristics (byte, packet, temporal and spatial) are the same as measured in routers in live environments. The parameters for creating the streams are extracted from packet traces or Netflow logs [1].

The design of Harpoon consists of 2 levels: the session level and connection level. The session level is made of 2 components: the number of active sessions and the IP spatial distribution. Using packet traces or Netflow logs an empirical distribution, $P_{ActiveSessions}$,is created which defines the average number of sessions (TCP/UDP) that are active at any point in time. The same way 2 other distributions are created $P_{IPRange_{src}}$ and $P_{IPRange_{dest}}$ which form the IP spatial distribution. For the lower layer, the connection level, 2 distributions are created: $P_{FileSize}$ and $P_{InterConnection}$. They define the size of a file during a transfer and the time interval for connection initiations respectively. The result of this is figure 2.1.

The biggest drawback of Harpoon is that it ignores the content of the payload. This makes it unsuitable for testing an IDS that operates on the application layer.

Trident

Trident is a traffic generation framework for online evaluations of NIDSs. It is based upon MACE (see Section 2.3.4) and Harpoon, both previous work of the authors. Trident is able, unlike many other traffic generators, to generate mixed traffic (malicious and benign traffic combined). It does so by using Harpoon for benign traffic generation and MACE for malicious traffic generation [2].

For this, the Trident system is designed, as can be seen in Figure 2.2 which represents the benign traffic generation part. First, traffic grooming is applied.

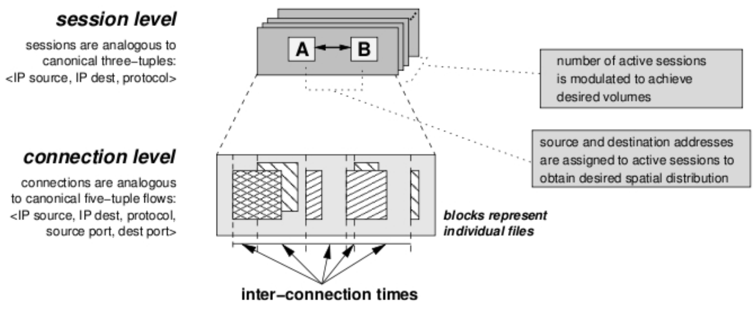
session level
sessions are analogous to canonical three–tuples: <IP source, IP dest, protocol>
A
B
number of active sessions is modulated to achieve desired volumes
source and destination addresses are assigned to active sessions to obtain desired spatial distribution
connection level
connections are analogous to canonical five–tuple flows: <IP source, IP dest, protocol, source port, dest port>
blocks represent individual files
inter–connection times

Figure 2.1: Harpoon's 2 level architecture. Figure taken from [1]

Here traffic is analyzed and categorised using a trust based strategy. Traffic is marked trusted, neutral or suspect, based on connection properties (such as destination, source, etc), behaviour properties (such as scanning), etc. In practice, traffic grooming faces the same problem as the IDSs: the correct classification of traffic. How traffic grooming works is outside the scope of this topic. The step after traffic grooming is payload classification. Here packets are classified into various pools that correspond to particular states in different service automata. In the Trident system, 2 service automata (SSH and HTTP) are implemented. After payload classification, payload sanitizing takes place. This process discards or modifies packets, such that they can be used for replaying. For the actual traffic generation new distributions are generated: $P_{NumPackets}$, $P_{FlowSize}$. They define the number of packets to send in each state, and the size of the total traffic to terminate the connection (when exceeded).

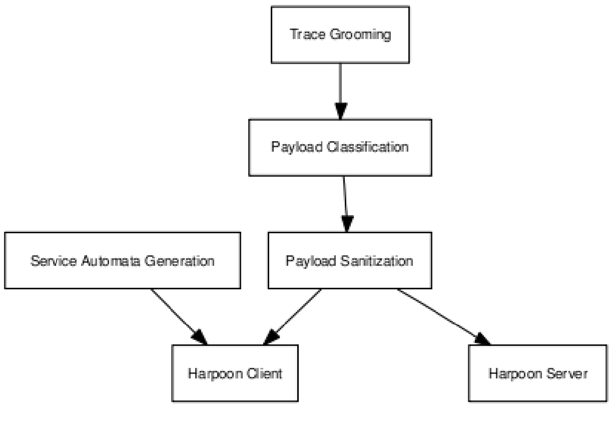

Figure 2.2: The Trident benign traffic generation system. Figure taken from [2]

Despite the size and dynamic parameters of the Trident system, it still lacks many important dynamic parameters. For example, the exploits with which MACE is enhanced do not contain any new exploits. SQL injection attacks are not included. Another problem is that only part of the payload is defined (the protocol state), no information on the parameters belonging to each state is given. Also the service automata are not complete, and only service automata for SSH and HTTP are (partly) implemented. The testing method used to test the performance of traffic grooming is flawed. Their algorithm is executed and the result is compared to the result of the same data analysed by Snort. Since Snort is a misuse based IDS, 0day exploits are ignored. Therefore no real statement can be made about the performance of the traffic grooming algorithm.

Luo's model

Very intensive research was conducted by Luo in his PhD thesis [20]. Despite the absence of a payload implementation his work is still valuable in potential since it contains detailed information about the relation between protocols and different probability distributions. Besides this, Luo uses Honest, a testbed for the evaluation of IDSs. The most detailed information on traffic generation in [21] is "When a session is generated, an expect script is called that creates an actual session for the selected protocol, and generates traffic on the data network". Unfortunately this is not detailed enough for a thorough analysis.

2.3.2 Data sets

Data sets have been the primary resource for IDS evaluation. They can contain both benign and malicious data. General problems with the use of data sets are for example whether or not real data is used, and if the data is sanitized or not. An advantage is that it can be analysed thoroughly. The rest of this section describes and comments on various data sets aimed at IDS evaluation.

DARPA data set

The DARPA data set is the most widely used data set for IDS testing. Two versions were created by the Lincoln Laboratory of MIT under DARPA funding in 1998 and 1999. For the creation of this set an Air Force base with thousands of machines and hundreds of users was simulated. Various scripts were used to create realistic traffic of several user types (programmers, managers, secretaries, etc).

First, traffic was analysed for the period of 4 months at Hanscom Air Force Base and 50 other bases. Statistics are collected from this data and used to create synthetic traffic. Figure 2.3 resembles the setup used to create the synthetic traffic for the '99 data set. As can be seen, a single host resembles many workstations or servers. For the creation of SMTP traffic, real e-mails from various mailing lists were used. A similar approach was used for FTP traffic. HTTP traffic was created slightly differently, by downloading and updating websites daily on the simulation network.

In [22] a comparison is made between the DARPA data set and fresh collected data (FIT). Although this focuses mainly on the lower level aspects of network traffic, some noteworthy differences have been found:

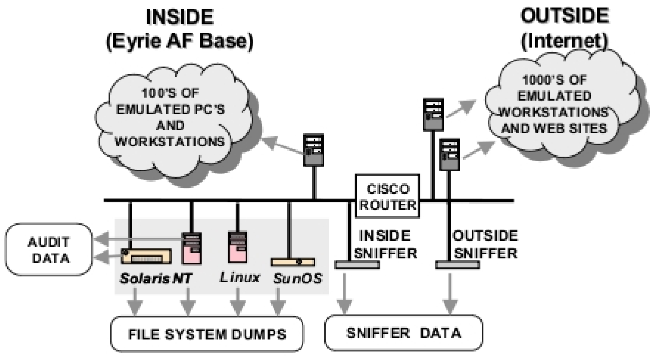

Figure 2.3: The setup to create the 1999 DARPA set. Figure taken from [3]

- In the DARPA data set there are only 29 distinct source addresses in the first and third week. This seems relatively low for an Air Force base.
- In the DARPA data set TCP SYN packets during week 1 and 3 contain always 4 option bytes. FIT has however 103 different values for the 4 option bytes.
- In the TTL field of the DARPA data set only 9 different values were observed. In FIT however 44 values were observed.
- The DARPA data set only contains GET requests in HTTP sessions, which is rather unlikely.
- The DARPA data set only uses 1 SSH client.

However the biggest problem with the data set is its outdatedness. There are relatively few and by now outdated attacks present. For example there are no SQL injection attacks.

KDD Cup data set

The KDD Cup is the annual data mining and knowledge discovery competition organised by the ACM Special Interest Group on Knowledge Discovery and Data Mining. For the 1999 contest, a classifier had to be taught to differentiate between legitimate and illegitimate connections. The training and test data were provided by Prof. Salvatore Stolfo of Columbia University and Prof Wenke Lee of North Carolina State University. This data set is used by researchers to test their IDS [23] [24].

However, this data set is based on the 1998 DARPA data set and therefore holds many of the same problems as the DARPA data set.

DEFCON data set

During the annual Defcon conference in Las Vegas the Shmoo security group logged all the traffic in 2000, 2002 and 2003 during the Capture The Flag (CTF) event. In this event hackers tried to hack several servers. As a consequence, the Defcon data set contains mostly malicious data but almost no background traffic. The advantages are that it contains real data, and real replies which are not based on any script[1]. Unfortunately, no in depth analysis has been made of the Defcon data set[25].

2.3.3 Frameworks

A framework can be seen as a collection of scriptable clients and one or more (scriptable) servers. The clients are responsible for traffic initiation, while the servers are used for sending genuine replies to the clients. A big advantage of frameworks over other testing methods is that traffic can be created dynamically, making it applicable for many test cases [19]. This is also the biggest drawback. Setting up a framework and creating the data can be time consuming, which is another drawback.

LARIAT

LARIAT is the abbreviation of "Lincoln Adaptable Real-time Information Assurance Testbed". It was developed in 2002 by Rossey et al. LARIAT is designed to support real-time evaluations and to create a deployable, configurable and user friendly testbed [4].

It is relatively user friendly and complete framework (in terms of services). To use it, a user selects a profile and one or more attack scenarios. The profile defines the services (FTP, SSH, etc) for which traffic is generated, the statistical distribution of each service and the traffic rate.

As defined by the Distribute Configuration phase, the LARIAT framework consists of several servers. This is depicted in Figure 2.4. The attack part of LARIAT consists of the following components:

Attack Component Abstraction and Management Each attack is accompanied by an XML file which describes all the properties of an attack.

Attack Framework API The API is used to launch attacks in their native format (Perl, binary, etc) and to store/retrieve information about attacks into a common data repository.

Attack Scenario Model This model links attack components together. For example if after a portscan a port has been found closed then no attack is launched on that port.

From this point traffic generation is automated which evolves in the following 6 phases:

1. The Initialize Network phase initializes the test network. Also the user accounts, log files and processes are set to pre-specified configurations.

2. The Distribute Configuration phase distributes the traffic configuration and the experiment details to the hosts.

3. The Pre-Conditions phase prepares the framework for the flow of synthetic traffic by, for example, building scripts for each background traffic session.

[1]Of course this does not hold for any type of scanner used during the competition

4. The Run Traffic phase actually creates/runs the traffic.

5. The Verify and Score phase tries to verify the successful completion of an attack by, for example, searching the victim host for evidence of the attack.

6. The Clean Up phase removes evidence of attacks, resets the changes made to the attack script, restores the hard disk from an image, etc.

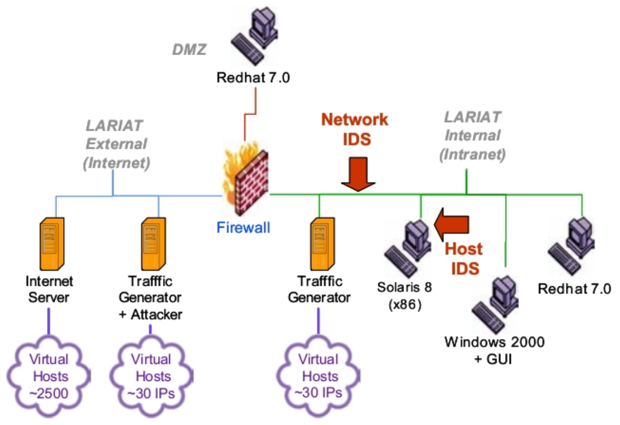

Figure 2.4: The LARIAT framework setup. Figure taken from [4]

This major drawback of the LARIAT framework is that it is not publicly available. Therefore no detailed information about the synthetic created background traffic is available. Another drawback is that the profiles are based on the DARPA data set, which is known to be outdated.

DETER

DETER is the abbreviation of the cyber DEfense Technology Experimental Research network. It is a closed testbed framework for the evaluation of security mechanisms. Access to the network is only allowed for researchers after they request an account. The testbed consists of around 1000 PCs and several commercial routers and switches. Currently 3 classes of attacks are supported [26]:

1. Denial Of Service attacks.

2. Worm attacks.

3. Routing/infrastructure attacks.

Several tools such as Stacheldraht, Trinoo (both for malicious traffic) and Harpoon, Webstone (for background traffic) are used for traffic generation. One of the problems with DETER is that no information is available on how exactly malicious traffic is generated. The creation of background traffic is also rather limited since Harpoon (discussed in Section 2.3.1) is used. Moreover SEER, a tool designed by DETER participants, contains serious limitations. For example, `wget` mirroring a website is used as the sole source to create both HTTP and FTP traffic [27] [28].

ViSe

ViSe is the abbreviation for Virtual Security Testbed. It is a framework containing 10 versions of popular operating systems with 40 exploits or programs against them. ViSe contains 3 classes of configured images [29]:

1. Attacker
2. Detector
3. Victim

Problem with ViSe is that no scripts or tools for background traffic are included, making it only usable to test signature based IDSs or anomaly based IDSs that were already fed with background traffic.

TIDeS

TIDeS is the abbreviation of Testbed for evaluating Intrusion Detection Systems. The testbed consists of 2 test scenarios [30]:

1. Non-environmental based, which does not depend on detected network data.
2. Environmental based testing scenarios.

The non-environmental scenario contains 3 scenarios:

- The All-Legitimate scenario only creates legitimate traffic.
- The All-Illegitimate scenario only create illegitimate traffic.
- The Mixed traffic scenario creates a mix of legitimate and illegitimate traffic.

TIDeS consists of 5 modular components:

1. The handler is the main controller and acts as an interface for the testbed. It interfaces with other components in the network and monitors tests.
2. The Virtual Machine Emulator is used to emulate various Operating Systems and machines on a single physical computer, thereby creating a virtual network.
3. The Launcher launches the traffic by activating the different scripts.
4. The Environment Profile Generator creates different profiles by analyzing traffic, for example a home environment profile and a university environment profile.
5. The scripts connect to a server and interact with its services, creating the traffic.

Although TIDeS specifies which exploits are used, it unfortunately does not contain any information on how the background traffic is generated, how the scripts work, etc. Therefore a good conclusion based upon the lower level functioning of TIDeS cannot be given since the details are not publicly available.

2.3.4 Attack mutation tools

Attack mutation tools are tools that mutate an exploit with the purpose of evading IDSs. Many signature based IDSs have specific signatures for exploits. Using an evasion technique or tool on an exploit can evade an IDS.

MACE

The Malicious trAffic Composition Environment (MACE) is a tool developed by Sommers et al for creating malicious traffic. The framework consists of 4 models that operate together [5]:

1. The Exploit Model contains a set of vulnerabilities.
2. The Obfuscation Model morphs the header or payload, such that the exploit evades a NIDS.
3. The Propagation Model defines the order in which the victims are to be attacked.
4. The Background Traffic Model creates benign traffic.

The models work together as depicted in figure 2.5, where the different models are depicted as jigsaw puzzle pieces. The attack vectors in this figure are attack profiles of which 4 are available (SYN flood, Welchia, Rose, Blaster).

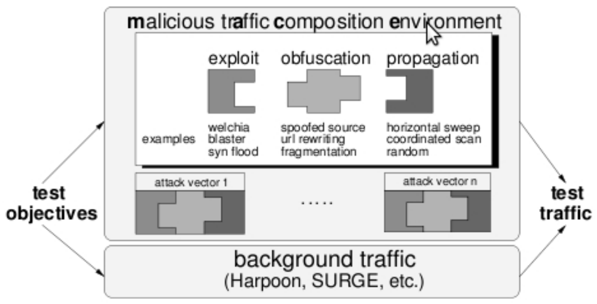

Figure 2.5: The MACE framework. Figure taken from [5]

MACE was tested by running 4 different types of attacks against a firewall and 2 IDSs. Results only focused on the performance of the firewall and IDSs and not on the evasion. Other drawbacks are that it is not publicly available and there are only 4 types of attacks implemented. The limited amount of attacks and the fact that exploits have to be ported to MACE code fragments (such that the obfuscation model can interpret it) makes MACE relatively unportable. All these drawbacks lead to the conclusion that MACE is not a suitable candidate to be used for IDS testing.

ADMmutate

ADMmutate is a shellcode mutation tool written by K2 (pseudonym of Shane Macaulay). It provides several techniques such as multiple code paths, non operational pad instructions, out-of-order decoder generation and randomly generated instructions. The shellcode is encoded with a function $E(S)$, where $E()$ is the encoding function and S the shellcode. This is encapsulated by the decoding function $D()$. The result is $D(E(S)) = S$ [31].

Although ADMmutate is an interesting tool, it lacks several important features such as support for architectures other than X86. More important is however that according to Robert Graham of Errata Security, modern IDSs do not even look at the shellcode. Instead the focus is on, for example, the triggering of a shell prompt. This makes ADMmutate not suitable for IDS testing [32].

Sploit

Sploit is an exploit mutation tool created by Balzarotti et. al. Just as with MACE an exploit has to be rewritten in a specific markup language, understandable by Sploit.

Sploit applies mutations on the network layer, application layer and exploit layer. To be precise, the following mutations are currently supported [33]:

- Using IPV6 instead of IPV4.
- IP packet splitting.
- Protocol rounds (run the exploit after 1 or more successful protocol rounds).
- FTP evasion techniques.
- HTTP evasion techniques (can also be applied to other protocols).
- SSL NULL Record Evasion.
- ADMmutate.
- Alternate encodings (such as UTF X).

Drawback of Sploit is that exploits have to be exported to a format interpretable by Sploit. Another drawback is that the set of mutation techniques is relatively limited.

The Metasploit framework

The Metasploit framework is a framework for exploit execution on remote targets. Exploits are written in a markup language, interpretable by the framework. The framework provides, depending on the exploit code of course, a list of possible IDS evasion techniques [34].

Drawback of this design is that exploits have to be exported to a markup language such that they can be used in the framework. Another problem, which makes it unsuitable for traffic generation, is that everything has to be selected by hand. There is no automated exploit mechanism implemented.

2.4 Conclusion

This chapter gave an introduction into IDSs, discussed IDS testing and discussed various types of input data. There was shown that in the worst case scenario each IDS, in each environment, for each test aim, needs its own input data when it is tested.

The input data is of great importance since the conclusion derived from testing stands or falls with the quality of the input data. Choosing the correct input data is thus of vital importance. In general there are some options available, each with its own advantages and drawbacks.

Tools for creating background are often quite simple. Characteristics of the payload are analyzed and used to create traffic. The actual payload at the application layer is ignored, which makes them unsuitable for proper IDS testing. A NIDS for example analyzes the payload at the application layer and detects that the payload is not according to the protocol specification.

Data sets have the advantage that they can be analyzed intensively. Also it is easy to compare IDSs with each other since the same input data is used. A drawback is that the currently available data sets are outdated. For example, in the DARPA set there are no SQL attacks. Data sets are also not easy to update and are not host specific. This makes them not ideal for IDS testing.

Many papers focus on IDS evasion techniques. It is noteworthy that the applied techniques are often well documented and tested. The discussed attack mutation tools often require the exploit to be converted to a certain script language which is then interpreted by the tool.

Frameworks have the big advantage that they are flexible in terms of creating traffic. Some even have the option to include attacks. A drawback is though, that the methods for benign traffic generation are often quite simplistic. Other drawbacks include scarce documentation, not being freely available, etc.

We notice that a well documented and well performing tool for benign traffic generation is missing. The focus from the scientific community in this area has been on IDS evasion techniques instead of benign traffic generation. Currently the best performing freely available framework is SEER. SEER simply creates benign traffic by running a program that mirrors a website/ftp server. This can be easily detected by an IDS that checks the client version and analyzes the think time between requests. Other frameworks such as LARIAT base their benign traffic generation on observed data such as e-mails from a mailing list. Unfortunately it is not open source and not well documented. Also, LARIAT contains flaws, since it was used to create the DARPA data set and is thus responsible for these flaws.

Because of these reasons we decide to make a tool for benign traffic generation. It analyzes several protocol runs as discussed in Chapter 3 and then uses this information to create benign traffic as discussed in Chapter 4.

Chapter 3

Protocol Analysis

This Chapter discusses the analysis of plain text application layer protocols with no prior knowledge of the protocol. This information is later used in Chapter 4 where a semi state machine is created.

The first Section, 3.1, gives a general introduction, followed by Section 3.2 which describes how raw packet dumps are transformed into protocol runs that are put in a database, containing requests and responses. Then, Section 3.3 describes how the various protocol delimiters are detected, after which Section 3.4 discusses how various identifiers and variables are detected. The section that follows, Section 3.5, discusses how tokens are classified into groups. Finally the last section, Section 3.6, discusses user sessions and which properties they have.

3.1 Introduction

Communication over the Internet is achieved by sending data over layers. Data is encapsulated when it moves one layer down. The added encapsulating data is removed, when it moves a layer up. This means that the highest layer, the application layer, receives the exact same data as has been sent when there is a successful exchange.

One of the consequences of this approach is that each layer requires its own stack. This implies that every layer contains a potential security threat. In the real world however exploits for layers below the protocol layer are rare. In fact, IBM's X-force states that in 2010 over 55% of the vulnerabilities in that year affect web applications[35]. This is a remarkably high number since Ipoque showed that around 25% of the total Internet traffic is HTTP traffic[36].

Other important information found in [36] is that HTTP is the most used protocol, followed by P2P, streaming and IM protocols. Epoque states furthermore that relatively old protocols such as NTP, IRC, SMTP are still in the top 20 of most used protocols. They are accompanied by more modern protocols such as bittorrent. Interesting to see is that most of the top 20 protocols are plain text, although some of them require the encryption layer or functionality to be removed, for instance HTTPS.

In 2001 RFC 3117 was written, which discusses the design of application layer protocols. Its importance lies in the fact that although it describes the BXXP protocol, it still gives valuable information on how to design an application layer protocol[37]. Many plain text protocols share the same properties, although the implementation can be different.

The rest of this chapter describes various techniques that use a few protocol runs in order to find these properties. Because of the fact that HTTP is the most used protocol, and most vulnerabilities are found in web applications, techniques that fit the HTTP protocol are used. These techniques can however, possibly with some modifications, be used for other plain text protocols as well.

3.2 Protocol runs

Protocol runs consist of requests and responses. Depending on the protocol the request and responses can be sent by any number of clients and servers. For example, in some P2P protocols both clients are able to send requests to each other, while in a request-response protocol requests are sent solely by the client and responses solely by the server.

When an unknown protocol is analyzed, it is difficult to use terms such as request and response. The result is that they can not be differentiated from each other without prior protocol knowledge. Therefore, we define a source IP address, which we assume to belong to the client. We assume the other IPs that the client has connectivity with, to be servers, even though they can be other clients in a non request-response protocol.

A protocol run starts when a client opens the first socket to a server. In our analysis we assume the first packet sent after establishing a connection, to be a request. Logically, the packets that are sent in reverse direction are marked as responses.

Using the above described techniques a `pcap` file, created by `tcpdump` is analyzed and parsed. All the requests and responses are put in a database, together with other characteristics such as source and destination IP, timestamps, etc. The database contains the requests and responses as they are parsed to the application that implements the application layer protocol and is used as a base to analyze the protocol as explained in the rest of this chapter.

3.3 Delimiters

According to [38], a delimiter is: “A character used to indicate the begin and end of a character string, i.e., a symbol stream, such as words, groups of words, or frames. 2. A flag that separates and organizes items of data”. A delimiter is used for framing: “which defines how the beginning and ending of each message is delimited”. Framing is a technique used by protocols [37].

Detecting delimiters is the first step in analyzing an unknown plain text protocol. Once the delimiters have been found, the protocol can be broken down and analyzed further. This is especially important for the semi state machines discussed in Chapter 4, since it needs to (partly) parse the protocol.

3.3.1 Detecting delimiters

In theory there is no limit on the delimiter size. In practise, however, mostly up to 5 byte combinations are used for the top 20 protocols mentioned in Section 3.1.

```
GET / HTTP/1.1\r\n
Host: www.voetbalzone.nl\r\n
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-gb)
AppleWebKit/533.17.8 KHTML, like Gecko) Version/5.0.1 Safari/533.17.8\r\n
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,
image/png,*/*;q=0.5\r\n
Accept-Language: en-gb\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n\r\n

GET /css/all.css HTTP/1.1\r\n
Host: www.voetbalzone.nl\r\n
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-gb)
AppleWebKit/533.17.8 (KHTML, like Gecko) Version/5.0.1 Safari/533.17.8\r\n
Accept: text/css,*/*;q=0.1\r\n
Referer: http://www.voetbalzone.nl/\r\n
Accept-Language: en-gb\r\n
Accept-Encoding: gzip, deflate\r\n
Cookie: ASPSESSIONIDQATTQSRD=MAOIACCDONEJEKMMKHBJEHEI; VZ%5Fcookies=1%09\r\n
Connection: keep-alive\r\n\r\n
```

Figure 3.1: Two HTTP headers. Note that the third and the fourth line, plus the fifth and the sixth line of the first header, as well as the third and the fourth of the second header are actually one line. Due to size limitations they appear as two lines.

Delimiters can occur, according to [38] at 3 places: between words, between groups of words and between frames. Figure 3.1 displays two sample HTTP headers. Here the word delimiter is a “ ” (the space character), the group delimiter is “\r\n” and the frame delimiter is “\r\n\r\n”.

3.3.2 Detecting frame delimiters

Detecting frame delimiters is fairly easy since they occur at the begin and end of a frame.

The pseudo Algorithm 1 illustrates how the ending frame delimiter is found. The same algorithm, with a small modification can be used to find the starting delimiter.

Algorithm 1 simply puts the last i characters of each request in a map. If the size of the map is equal to 1, then it means all the requests contained the same delimiter and the delimiter thus has been found. By starting with searching for the largest delimiter, it is made sure that the right delimiter has been found.

In the case of HTTP it is shown, when Figure 3.1 is used as input, that the correct delimiter is found when $n \geq 4$.

```
Input: A group of requests Requests, the maximum size of the to be
       searched for delimiter size
Output: The ending delimiter del, if found
Data: A map m, that holds the delimiters and how many times they
      have been found, an integer i to denote the size, a request r.

foreach $i \leftarrow size$ to 1 do
    foreach $r \in Requests$ do
        add_delimiter_to_map(m, r.pkt, i);
    end
    if m.size = 1 AND del = NULL then
        del ← create_delimiter_from_map(m);
        break;
    end
end
return del;
```

Algorithm 1: `Find_Frame_Delimiter`(R)

3.3.3 Group delimiter

Group delimiters can appear throughout the whole request. The difficulty lies in detecting the delimiters since the size is unknown. In theory it can be any character or sequence of characters. Fortunately, based on an assumption it is still possible to find the delimiter.

For example, when protocols are designed the delimiters are often chosen as a sequence of characters, or a character that does not appear in the text too often. This to overcome excessive use of escape characters. Logically, a sequence of rarely used characters is less likely to occur then just one character.

Using this assumption we created Algorithm 2. It maps all possible n-grams in a request and how many times this n-gram occurred. The n-gram with the most combinations is the most likely delimiter.

```
Input: A group of requests Requests, the maximum size of the to be
       searched for delimiter size
Output: The group delimiter del, if found
Data: A dynamic sized FIFO buffer buf of size i and a map m, two
      integers $i, j$ used as the size and index respectively, a request r.

foreach $i \leftarrow size$ to 1 do
    foreach $r \in Requests$ do
        foreach $j \leftarrow 0$ to r.len do
            add_character_to_buf(buf, r[j]);
            if buf_is_full(buf) then
                add_buf_to_map(buf, m);
            end
        end
    end
end
del ← create_delimiter(map);
```

Algorithm 2: `Find_Group_Delimiter`(R)

In the case of HTTP, when Algorithm 2 is applied on the headers found in Figure 3.1, the outcome is the delimiter “\r\n”.

3.3.4 Word delimiter

Detecting the word delimiter takes a different approach from detecting the other two types of delimiters, based on the following assumption:

- If there is data between any combination of frame and/or group delimiter, then the data contains the word delimiter.

The assumption is based on the idea that there is always more than 1 item in a group. In order to find the delimiter, the first group is tokenized and all the n-grams are put in a map. After this, all the groups of tokens of all requests are searched for each item in the map. If an item from the map does not occur in a group, then that item is removed from the map and the process of searching for an item from the map in a group restarts. If everything went well then the map contains only one item: the delimiter.

This results in Algorithm 3.

In the case of HTTP, when Algorithm 3 is applied on the headers found in Figure 3.1, the outcome is the delimiter “ ” (the space character).

3.4 Identifiers and variables

Identifiers and variables play an important part in application layer protocols. They play a pivotal role in protocol runs since they contain important information that can be session, user, client, etc, specific. For this reason it is important to correctly identify them.

The first step is tokenize all requests for all sessions. For each unique token t_n, an object o, is created. Each o contains a session object so, for each session t_n was found in. Every so contains a map and a counter, which specifies the number of times t_n was found in the session. The map holds the tokens t_{n+1} that follow t_n in that session, and the number of times t_n is found.

The first type of identifiers are found by calculating for each o the total number of times t_n is found, divided by the number of unique tokens following t_n. If this value is above a certain threshold then t_n is defined as an identifier and all the t_{n+1} as variables.

Protocols can also have variables and identifiers that are session specific. This includes for example the client version specification. These types of identifiers and variables are marked as client variables and client identifiers. They are found by verifying whether t_n holds in every so only one item (t_{n+1}). If this holds, then t_n is marked as a client/protocol identifiers and the t_{n+1} as client/protocol variables.

In the case of HTTP this algorithm will mark each first token in a request as an identifier and each second token in a request as a variable. Which is logical since the second token is often the file requested, information posted, etc and typically varies a lot. This information is useful when several sessions are mixed into a new session (see Section 4.4).

```
Input: A group of requests $Requests$, the maximum size of the to be
        searched for delimiter $size$, the group delimiter
Output: The item delimiter $del$, if found
Data: A dynamic sized FIFO buffer $buf$ of size $i$, a map $m$, a group $g$
       obtained by using the group delimiter, a request $r$
$r \leftarrow$ get_first_request(Requests);
$g \leftarrow$ get_first_group(r);
foreach $i \leftarrow size$ to 1 do
    foreach $j \leftarrow g.len$ to 1 do
        add_character_to_buf(buf, g[j]);
        if buf_is_full(buf) then
            add_ngram(buf, m;
        end
    end
end
next;
foreach $buf \in m$ do
    foreach $r \in Requests$ do
        while $g \leftarrow$ tokenize_group(r) do
            if is_not_in_group(buf, g) then
                remove_buf_from_map(buf, m);
                goto next;
            end
        end
    end
end
if m.size = 1 then
    $del \leftarrow$ create_delimiter_from_map(m);
end
return del;
```
Algorithm 3: `Find_Item_Delimiter`(R)

3.5 Token types

The previous section already classified tokens into identifiers, variables and protocol/client defined identifiers and variables. Protocols however can also have protocol/client defined tokens and user defined tokens. Also tokens can be dynamic, meaning there is a relation between the token in the request and a token received earlier. The rest of this section explains how tokens are classified in the aforementioned categories.

3.5.1 Protocol/client defined tokens

In our thesis we define protocol/client tokens as identifiers and certain variables that are determined in the protocol specification or by the client program. Thus not by the user.

A few assumptions are made in order to determine protocol/client defined tokens.

- The first request only contains protocol/client defined tokens except for the tokens that are defined as variables.
- All identifiers are protocol/client defined tokens.

Specifying protocol/client defined tokens is useful, since it becomes clear for which tokens it is required to search for a relation with a previous received response, as discussed in Section 3.5.2.

3.5.2 Dynamic tokens

In order to create artificial traffic as proposed in Section 1.2, requests must be created. Often tokens in the request are related to tokens in the response. To be more precise, a token or part of a token from a request req_t did not occur before in a request, but is found (possibly partly) in a response res_t such that $res\_t_x < req\_t_y$ where x and y are points in time. As mentioned res_t and req_t can partly differ from each other. For example the first and the last character of req_t do not match with res_t or part of req_t (where again the first and last character can differ) is a sub string of res_t.

Of course not all tokens need to be analyzed. For the protocol/client defined tokens a relation does not need to be searched for since these are defined by the protocol or client. Variables on the other hand can depend on previously received responses. For example in the case of HTTP, the token that specifies the file being requested can be found in a previously received response. Just as well as the token that identifies the cookie.

When such a relation is found a dynamic token is created, for which the following constants are used:

res_t_{start}	The first character of res_t.
req_t_{start}	The first character of req_t.
$res\_t_{match\_start}$	The point in res_t where res_t and req_t start to (partly) match.
$req\_t_{match\_start}$	The point in req_t where res_t and req_t start to (partly) match.

$res\_t_{match\_end}$ The point in res_t where res_t and req_t do not match anymore.

$req\_t_{match\_end}$ The point in req_t where res_t and req_t do not match anymore.

req_t_{end} The end of req_t.

res_t_{end} The end of res_t.

$match_{length}$ The length of the match.

req_t_{length} The length of the request.

res_t_{length} The length of the response.

A dynamic token holds the following properties:

Request ending delimiter The delimiter that comes after req_t. This delimiter is used for future traffic generation.

Request starting character When the first character of res_t and req_t do not match, this is set to req_t_{start}.

Request ending character When $req\_t_{length} - match_{length} = 1$ and the request starting character is set, then it is set to req_t_{end}. When $req\_t_{length} - match_{length} = 2$, then it is also set to req_t_{end}.

Response starting string When $res\_t_{match\_start} > res\_t_{start}$ then this is set to its difference.

Response ending character When $res\_t_{match\_end}$ does not match res_t_{end} and there is a character after $res\_t_{match\_end}$, then this is set to that character.

Offset When response starting string and/or response ending character are set, then the offset represents the n^{th} time one of these or a combination of these occurs in res. When these are not set, the offset specifies the offset of the token in terms of tokens in res.

3.5.3 User defined tokens.

Certain plain text protocols such as IM protocols depend on user generated input. This input can be in a certain natural language, often with certain limitations such as the length of the input. Detecting whether a language is used, when a group of words is given as input is possible [39][40].

The algorithm requires groups of tokens. Groups of tokens are parsed using the group delimiter found in 3.3.3. Of course not every group of tokens can be used. It must contain at least one token that is not yet defined. Since identifiers are defined by the protocol, they are not used in the algorithm. If a natural language is found, then the following statistics are created:

- The natural language.
- The amount of tokens found.
- The lengths of the tokens.

- The minimum amount of bytes found in a group.
- The maximum amount of bytes found in a group.

In the case of HTTP user defined tokens are, for example, the second token in a request when the user submits a message to an Internet forum.

All the tokens in that group that were not defined are now marked as user defined.

3.5.4 Undefined tokens

All the tokens that were not defined are marked as undefined tokens. These tokens are static, since did they not occur before and therefore, the only property that they hold is the token itself.

3.6 Client profiles

It is possible that there are different client programs/versions available for a certain protocol. Assuming this, it is also possible that these different programs/versions behave different from one another. When traffic is generated different programs/versions should not appear in the same session. To fulfil these requirements a client profile algorithm is defined.

The algorithm assumes the following two options:

- Some variables can occur with all client programs/versions.
- Some variables are client program/version specific.

Furthermore a variable can be a file. A file is defined, in this thesis, as a string containing a dot. In old operating systems such as CP/M and MS-DOS support files, here a file was made up of a base name and the file name extension separated by a dot. These old operating systems however required file names not to have an extension of more than 3 characters, while the minimum was 2. As a result these characteristics are still found in many files these days. However the length of the extension is not limited to 2 or 3 characters, especially in POSIX environments where file names such as .bzip2 occur.

For each session, the variables as defined in Section 3.4, are stored per request. Each variable is then checked whether it is a file or not. Finally all the found file types are saved. All the variables that are non file types are also saved, making each session have a list of supported file types and variables. This information, together with the in Section 3.5.1 defined information form the client profile. In the case of HTTP, client profiles are useful for constructing a semi state machine (see Section 4.5).

Chapter 4

Mixing

This chapter discusses various techniques to create a semi state machine for the HTTP protocol, based on the information obtained in Chapter 3[1]. The semi state machine comes in the form of C program code, that is able to, once compiled, setup a connection to a server, send requests and receive responses.

First, an introduction is given in Section 4.1, followed by Section 4.2 which discusses states, how they are used, etc. Then various statistical distributions are discussed in Section 4.3. The next section, Section 4.4, discusses how a semi state machine is generated. The last section, Section 4.6, is dedicated to the program that is used to generate traffic.

4.1 Introduction

Artificial traffic can be based on non artificial traffic, (see Section 2.3.2) and can be completely artificial, often containing random data (see Section 2.3.1). The goal of the proposed techniques in this chapter and Chapter 3 are to create artificial traffic based on non artificial traffic. The to be created artificial traffic is primarily based on data found in the application layer. Since this layer implements the protocol specification, it can be, when known, parsed relatively fast and easy.

According to [41] a compiler is: "a program that can read a program in one language - the source language - and can translate it into an equivalent program in another language - the target language". Application layer protocols are not programs that need to be compiled. Implementing an application layer protocol can however use several techniques from compilers: parsing and state machines. For both parsing and state machine creation there are several techniques available, such as LL and LR parsing for the parsing part and Deterministic and Non Deterministic finite automata for state machines. Which technique to use can depend on the protocol specification as well as the programmer's personal preferences.

[1]Although it is not a real HTTP state machine, since a real state machine considers error codes, it is still a semi state machine since states are defined differently (see Section 4.2)

4.2 States

We define a state as a template representing a request. A state is identified by the second token in the request, which is typically the file being requested. The template is created when the semi state machine is created (see Section 4.4), and contains the information from Section 3.5, so it is known how to construct a request.

A state can be, just like a token, protocol/client defined or user defined. A user defined state is a request for a file, or posted information, etc, that is specified by the user. For example, in the case of HTTP, it can be a request from a user when she clicks on a link. Protocol client defined states on the other hand are requests done by the client program without direct interference of the end user. For example, when a user loads a webpage, all the related content such as CSS files, jpg files, etc, are protocol/client defined. They are found by comparing the time between the last received response and the last sent request. This time is denoted by Δt. When Δt exceeds a certain threshold, then the last request is marked as a user defined state. If it does not exceed the threshold then the last request is marked as a protocol/client defined state.

States can also depend on other states, just like dynamic tokens as defined in Section 3.5.2. It is possible that a state can depend on multiple other states. For example, in the case of HTTP, this can happen when a picture is included on multiple pages.

4.3 Statistics

Important aspects for traffic generation include, but are not limited to, the number of user states to visit during a session, the think time between the users request and the last received response. These aspects can be determined by creating statistics, obtained from the original sample sessions. They are useful for traffic generation (see Section 4.4).

4.3.1 Related work

In [20] various statistics concerning HTTP are discussed. Although it seems promising, it still has some drawbacks such as pages that are identified by setting a threshold, not by an analysis of the protocol. This can be invalid when applied to the sample sets used in Chapter 5. Also the distribution for the number of user pages is based on two datasets, which differ from the one used in this thesis. This makes [20] not usable for this thesis.

4.3.2 Statistical distributions

Creating proper statistics for the properties mentioned in Section 4.3 is of vital importance for traffic generation. One solution is to assume a certain distribution based on prior protocol knowledge and statistics. For HTTP one expects the number of user states to be a Gaussian distribution. However a Gaussian distribution is primarily used to calculate $P(X)$, which is the probability of X given a certain distribution with possible variables set. The solution is to calculate X given a certain $P(X)$ value, where $P(X)$ can be determined randomly.

The biggest problem with assuming a Gaussian distribution is determining the credibility of the mean value μ and calculating the variance σ^2.

With a limited sample set, for example size 2, the following assumptions are made when using a Gaussian distribution:

- There is a Gaussian distribution
- The mean value $E(x)$, is the most likely value to occur, although (depending on the sample set), this value did not occur at all in the sample set.

These assumptions are important since they influence the to be created traffic. Unfortunately they are not based on facts. Also a problem with the Gaussian distribution is that extremely high and low values are possible, even though the chance of this to happen is really small.

4.3.3 Kernel density estimation

When statistics need to be created from a limited size set, a histogram can be made. Intervals are chosen in which the data from the set falls. A problem with a histogram is that by definition it is discrete, while the observed data can be generated by a continous function.

A kernel histogram can be used to overcome this problem. Here, every data point is associated with a function called the kernel function. These kernel functions depend on data itself, referred to as the bandwith. It is possible to use different functions, however, the most used one is a Gaussian function.

A big challenge with choosing the bandwith is to determine the optimal bandwith. When the bandwith is too small, the result will be "spiky", while when the bandwith is too wide, it will be hard to interpret the function. There are however several techniques available, as can be found in [6], to set a proper bandwith. Figure 4.1 shows how different bandwiths affect the result.

4.4 Differentiating

Using kernel density estimation with as input the number of visited user states in each session, the probability density function is calculated. As a bandwith selector the common variation, as defined in [42], is used. Using the Mersenne prime PRNG a number is generated and used to obtain a value from the probability density fucntion. The resulting number is used as the number of user states to visit. In the rest of this chapter, selecting random means obtaining a value as described above.

The first step is to randomly select a client profile as defined in Section 3.6. According to this profile a starting state is selected. The next step is to select the protocol/client defined states. These are randomly picked according to the user profile. This results, in the case of HTTP, to the request of certain files which are set in the user profile. For example, a client profile that does not support javascript does not request javascript files and the files associated with it.

After all the appropiate protocol/client defined states are selected, the think time is selected randomly. Next, another user state is selected, and all the

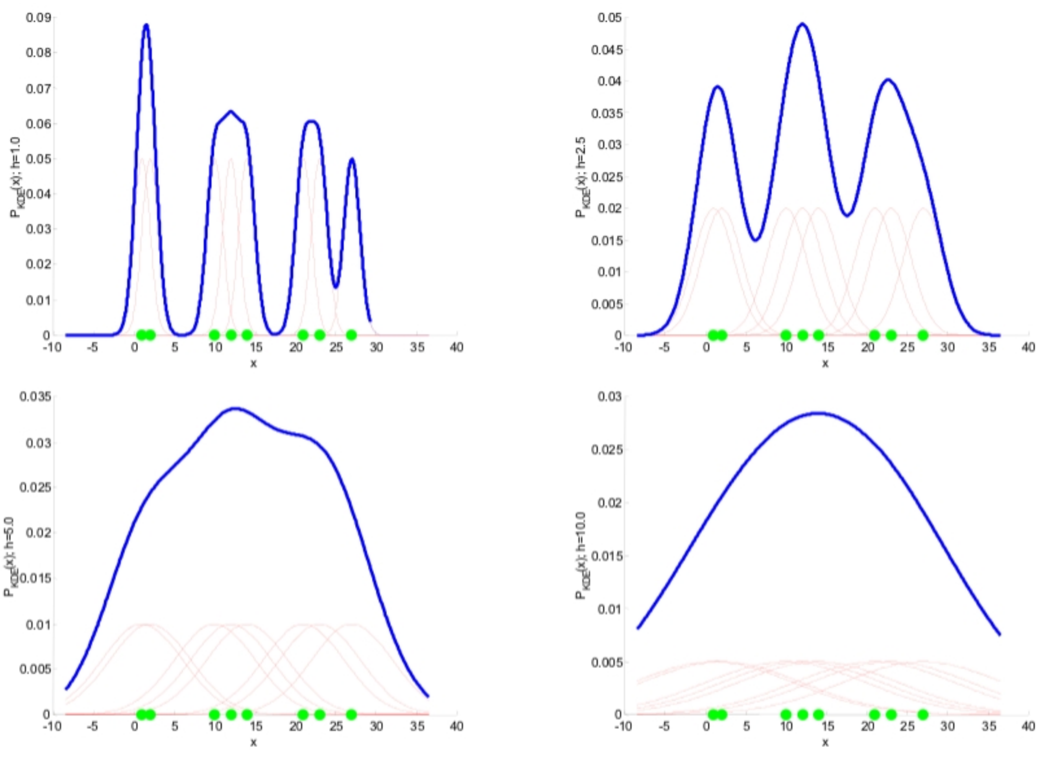

Figure 4.1: Different bandwiths for the same kernel. Image taken from [6]

protocol/client states that have not been visited are visited. At this point it is possible that certain protocol/client states have been accessed before, which is not realistic behaviour. Therefore, the protocol/client states that are known to be accessed more than once are accessed again, if they occur in that user state.

This way a semi state machine is created with a defined transition trace.

4.5 Traffic generation

A problem with states is that not all states occur in a certain client profile, making it difficult to use the template since essential information is missing. An example is that client A visits a user state U, while client B does not. After the data has been mixed, a state transition trace based on client B, wants to visit state U. This scenario is likely when the sample sets are limited and are of limited size.

To overcome this problem the following assumption is made:

- The type of file requested determines the content of the request.

Based on this assumption the chances of creating a proper request increases. To be more precise: when a certain file type was requested before, then this request is used as a template, with the exception of the file requested.

However, it is possible that the request is for a non-file type. At this point, for each group in the template, as defined in Section 3.3.3, the most likely group

Figure 4.2: Two HTTP headers. Again due to space limitations some lines appear as 2 or more lines while they are in fact one line.

```
GET /momkai/core/js/external/x.js HTTP/1.1\r\n
Host: www.voetbalzone.nl\r\n
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-gb)\r\n
Accept: */*\r\n
Referer: http://www.voetbalzone.nl/\r\n
Accept-Language: en-gb\r\n
Accept-Encoding: gzip, deflate\r\n
Cookie: ASPSESSIONIDQATTQSRD=MAOIACCDONEJEKMMKHBJEHEI; VZ%5Fcookies=1%09\r\n
Connection: keep-alive\r\n\r\n

GET /momkai/core/js/external/swfobject_2_2.js HTTP/1.1\r\n
Host: www.voetbalzone.nl\r\n
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.10)\r\n
Accept: */*\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
Referer: http://www.voetbalzone.nl/\r\n
Cookie: __utma=9678258.346101969.1285102837.1285102837.1285102837.1;
__utmz=9678258.1285102837.1.1.utmccn=(direct)|utmcsr=(direct)|utmcmd=(none);
__gads=ID=b1db2a927f14c628:T=1285102837:S=ALNI_Maj4v4BMr-DzVIqIxPpUeRykEzoCA;
pcid=C479D464-AE90-0001-60D1-198810FC24E0; VZ%5Fcookies=1%09;
ASPSESSIONIDQATTQSRD=NLDGACCDLMEIDABENMBDBOHD\r\n
Connection: keep-alive\r\n\r\n
```

is determined, based on groups from previous requests for which a template was previously successfully created. A group match comparison is used that calculates the comparison between the tokens in the group. The group with the highest comparison is the group to be used in the request.

The group matching algorithm is fairly simple. If a token matches it is given a *value* of 1. If it partly matches, then $0 \leq value < 1$ holds. The sum of the matches is the outcome of the algorithm.

Figure 4.5 shows two HTTP requests for different files. The requests differ quite a bit, partly due to the fact that they are requested with a different browser. Say our tool tries to create a template for the first HTTP header, while this header is not known (in contrary to what is shown here), and that only the second header is known. The algorithm then tries to find a matching line for, for example, "User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_4; en-gb)". The line from the second request which matches the most with this line is: "User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.10)". This line is then chosen to appear in the request.

4.6 The traffic generation program

The semi state machine comes in the form of a C program. There is chosen to use the `libivykis` library for the communication between the client and the server. This library is a small wrapper that can be used with various OS's, making our program usable on a variety of OS's. A drawback is that there is less control over various socket functions such as `setsockopt` in the posix environment.

As mentioned in Section 4.2, each state represents a request, and requests can depend on previous requests (Section 3.5.2). From this, it is quite easy to see which responses need to be parsed in order to be able to construct other requests. For example, in the case of HTTP, a response from a request for an image does not need to be parsed, while a response from a request for a CSS file likely needs to be parsed. To speed parsing up, only the relevant responses are parsed.

The socket behavior of our tool is modelled after the behavior of the proxy server, which is defined as "Don't open a connection if there are already enough in progress, except if the server doesn't do persistent connections and there's only one in progress" [43]. This in order to make our tool behave as realistic as possible.

It is possible that a token required to construct a request cannot be found, because the response changed a lot in comparison to when the sample sessions were analyzed. In such a case the request is dropped and the next request is sent.

Chapter 5

Testing

This chapter discusses the testing of our tool. Recall that the objective of our tool is to create benign artificial traffic, better than is currently possible (`wget -m`). Section 1.3 already stated that our tool consists of two parts: A protocol analysis part discussed in Chapter 3 and a traffic generation part, discussed in Chapter 4. The traffic generation part uses information obtained in the protocol analysis part. This implies that testing the traffic generation part includes testing the protocol analysis part.

The aim of testing is to obtain answers to several questions. For example, "Does it work?" and "How well does it work?". The first question is usually answered during the development process. If it does not work then there is a bug that needs to be fixed. The second question is more interesting and is answered in this chapter.

The rest of this chapter describes the testing of our tool. First, the test setup is described in Section 5.1, followed by Section 5.2 on test cases. Finally, Section 5.3 yields the test results.

5.1 Test setup

This section describes several aspects of our test setup. First, in Section 5.1.1 is described how the sample sets are obtained, after which Section 5.1.2 describes the setup of several IDSs used for testing our tool. Next is Section 5.1.3, that describes the client programs that were used to create the sample sets and if applicable, any additional information. Finally, Section 5.1.4 states the various configuration parameters that were used in the protocol analysis part.

5.1.1 Proxy/NAT setup

Obtaining sample sets from real users can be a daunting task, especially because of privacy issues. In our case, as we want to test our tool for several websites see Section 5.2, it becomes even more complicated since we do not have control over the last hop the packets visit before arriving at the server.

In general there are 2 options to obtain sample data. The first, is to obtain data directly from the user's workstation. The second, is to reroute the user's traffic to a node we control and obtain it from there. The advantage of the first approach is that there is non altered user data. Unfortunately, from our experience, there are not many users who are able to (within a reasonable amount of time) install, configure and run a sniffer on their PC in order to save the right data. Writing a program for this is also a time consuming task. The advantage of the second approach is that most users are able to configure, for example, a proxy server in their browser. A drawback is that the data does not resemble the (socket) behavior of the client program but that of the proxy server.

We choose the second option. The reason for this is that users were not able to configure a sniffer on their workstation.

Figure 5.1 illustrates the test setup:

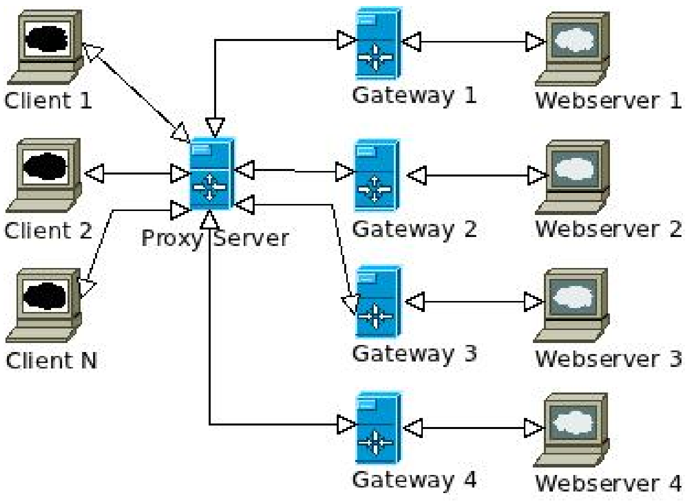

Figure 5.1: The test setup

We want to obtain sample sets for four different web servers. Therefore we set up a proxy server. This proxy server accepts up to n different clients, where n is a number determined by the proxy server setup. Once traffic arrives at the proxy server, it is, depending on the destination, rerouted to a gateway. To be more precise, all the traffic falling within the /24 network subnet class of the destination IP address, is rerouted to a gateway. Of course this only applies to the case where the destination IP address falls within the range of one of the four web servers we want to obtain sample sets from. The last step is to apply Network Address Translation on the gateway.

The advantage of this setup is that the added data by the client program (in order to connect through a proxy server to a website) is removed when it arrives at the gateway. Drawback is though, that while creating the sample set, only one user a time is allowed to visit a specific website. If this is not obeyed then it is difficult to differentiate between users. For the aforementioned reasons we decide to collect the data at the gateway from one user a time. The sample sessions obtained in this way, as well as the traffic generated by `wget` are saved in the `pcap` file format.

5.1.2 IDSs configuration

The rest of this section describes the various IDSs used and their setup.

Snort

Snort is a well known misuse based IDS for which quite some rules are available. We use the standard Snort installation with rules found at [44] and [45]. Some rules gave errors, but were fixed. No additional modification or configuration was applied.

Bro

Bro is a NIDS developed at the Lawrence Berkeley National Lab and the International Computer Science Institute. It ships standard with a collection of scripts that are used to detect certain intrusions. Bro also features a relatively advanced scripting language, especially in comparison to Snort.

We have developed a script (see Appendix A) that is able to detect `wget -m` by matching against the "User-Agent" field in the HTTP header and by having a counter set to a certain threshold. The counter holds the number of requests from a client. If the threshold is exceeded or the "User-Agent" fields matches, an alarm is raised. The value of the threshold is set by conducting empirical research on several `pcap` files of `wget` sessions.

Suricata

Suricata is an IDS written by the Open Information Security Foundation. It is able to use Snort rules and also provides an HTTP normalizer and parser. We use the standard setup to test our data. No additional Snort rules are added since these are already handled in our setup by Snort. The primary reason for using Suricata is the HTTP normalizer and parser since this specific work is not included in any other IDS.

5.1.3 Client programs

Table 5.1 shows the client versions that are used to create the sample sets and additional information on them.

5.1.4 Configuration parameters

Table 5.2 states the various used parameters that are used throughout this thesis:

5.2 Test cases

Six different users, each with their own Internet connection, were asked to configure their computer in such a way that the HTTP protocol uses a proxy server. They were asked to visit four different websites: `http://www.voetbalzone.nl`, `http://tweakers.net`, `http://nos.nl` and `http://www.security.nl`. These websites are updated daily. The websites were visited (via the proxy server) during a time interval of 5 hours. Traffic was recorded for each user, for each

Client	Version	Settings
1	Mozilla/5.0 (X11; Linux x86_64; rv:2.0) Gecko/20100101 Firefox/4.0	FireFox 4 running on Ubuntu 11.04 with Adblocks installed
2	Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:2.0.1) Gecko/20100101 Firefox/4.0.1	Firefox 4.0.1 running on Snow Leopard with Adblock plus, certificate patrol, ghostery
3	Mozilla/5.0 (X11; Linux i686; rv:2.0.1) Gecko/20100101 Firefox/4.0.1	FireFox 4 running on Ubuntu 11.04 with Adblocks installed
4	Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1	FireFox 4.0.1 running on Windows 7 with Adblock installed
5	Mozilla/5.0 (Windows NT 6.1) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.68 Safari/534.24	Google Chrome 11.0.696.68 running on Windows 7 with no additional filters installed
6	Mozilla/5.0 (X11; Linux i686; rv:2.0.1) Gecko/20100101 Firefox/4.0.1	FireFox 4.0.1 running on Ubuntu 11.04 with no additional filters installed

Table 5.1: The different client versions with their settings used to obtain our sample set

Algorithm	Value
Algorithm 1: Find_Frame_Delimiter; *size*:	6
Algorithm 2: Find_Group_Delimiter; *size*:	6
Algorithm 3: Find_Item_Delimiter; *size*:	6
Identifiers and variables (Section 3.4): Threshold value:	0.9
States (Section 4.2): Threshold value (Δt):	2700000 microseconds
Threshold value for the Bro `wget` detection script (Section 5.1.2):	400

Table 5.2: The various parameters used throughout this thesis

website. Also no users were visiting the same website at the same time, making each sample set contain data from only one user. The results are 24 unique sample sets.

The sample sets are used by our tool to create various executable programs as can be found in Chapter 4. The programs were started 2 days later. This time however, no proxy server was used. This way it is possible to test if our tool behaves in the same was a proxy server. If it would connect through a proxy server then this is not possible. The traffic generated by our tool is used as input for the IDS as described in Section 5.1.2.

5.3 Test results

This section is divided into three parts. The first part, Section 5.3.1, discusses the results of the test cases defined in Section 5.2. The second part, Section 5.3.2, gives an in depth analysis of the created traffic by our tool. Finally, Section 5.3.3 discusses statistics about our sample set and our artificial data.

5.3.1 Test case results

Table 5.3, 5.4, 5.5 and 5.6 show the outcome of the test cases as defined in Section 5.2. The number in the client column corresponds to the clients from

Client	Snort	Suricata	Bro
1	6 Denial of service alerts		
2	2 Denial of service alerts		
3	5 Denial of service alerts		
4	2 Denial of service alerts		
5	3 Denial of service alerts		
6	1 Denial of service alert		
Tool1	2 Denial of service alerts		
Tool2			
Tool3			
Tool4	1 Denial of service alert		
Tool5	4 Denial of service alerts		
Wget	1 Web-misc alert 7 Web crawl alerts 14 Denial of service alerts		Wget alert

Table 5.3: The responses of the IDS on the sample set, `wget` and our created data for `http://www.voetbalzone.nl`

Table 5.1. In the same column, tool1, tool2, tool3, tool4 and tool5 refer to our traffic generation program.

The tables contain short descriptions of the raised alerts. More information on these alerts is found in Appendix B.

Section 5.2 already mentioned that our tool does not connect through a proxy. The source IP however appears, according to Snort, from a malicious host. These rules can be safely ignored since they do not state anything about the quality of our tool and are therefore omitted from the tables in this section.

From Table 5.3, 5.4, 5.5 and 5.6 we see that Snort raises alerts for our tool, `wget` and our sample data. A description of the alerts is given in Appendix B.

The denial of service alert, specified in Appendix B.1, is a badly written rule and can thus be ignored. The website `http://www.security.nl` contains unicode encoding. The alarms raised considering unicode encoding can thus be ignored. More interesting is the website `http://tweakers.net`, where our sample sets raise alarms too. These are raised because more than 1 request appears in a single packet and because requests contain the string “archie”.

In short the alerts raised by our program, and our sample set can be explained and are false positives[1]. On the other hand traffic generated by `wget` is detected by Snort. In fact, there is detected that the website is crawled.

Suricata did not detect anything just as Bro. Bro however detected, with our script, that `wget` is used as a client.

5.3.2 In depth analysis

Even though the IDSs do not raise alarms upon data created by our tool, it does not mean our created data is realistic. A more in depth analysis of the traffic brought up several issues which are caused by the design of our tool. They can

[1]Although a misuse based IDS cannot raise false positives by definition, false positives still appear because of badly written rules

Client	Snort	Suricata	Bro
1	1 IIS unicode alert		
2	1 IIS unicode alert		
	1 Denial of service alert		
3	2 Denial of service alerts		
4	2 Denial of service alerts		
5	4 Denial of service alerts		
6	1 IIS unicode alert		
Tool1			
Tool2			
Tool3	1 Denial of service alert		
Tool4			
Tool5			
Wget	2 Web-misc alerts		Wget alert
	6 IIS unicode alerts		
	10 Denial of service alerts		
	5 Web crawl alerts		
	54 Double decoding alerts		

Table 5.4: The responses of the IDS on the sample set, `wget` and our created data for `http://www.security.nl`

be categorised in two categories: “malformed” requests and more fundamental design issues which do not lead to “malformed” requests.

Malformed requests

Malformed requests appear throughout our created data. For example, the item requested is malformed. This can happen when the layout of the page, in which the item appears, changes. For example, in the sample set the file “/image.jpg” is requested. In the sample set this file is encapsulated as follows:

```
src="image.jpg">
```

Part of the information on how to construct a proper request at a future point in time, is the request starting character, as defined in Section 3.5.2, which is set in this scenario to “/”. At a future point in time, it is possible that the response changes and by that the file requested. It becomes, for example:

```
src="http://example.com/image.jpg">
```

The consequence is that the requested file, sent by our tool becomes:
“/http://host.com/image.jpg”, which is obviously a malformed request.

Another is issue that causes a malformed request is a malformed “Host” field. In an HTTP request the “Host” field is important, since it helps the webserver to differentiate between different hosts when multiple websites are hosted on one server. When a file is requested from another server this field changes. It is possible that more or less the previous scenario applies, meaning that the host field can become invalid.

Another problem is that a template for a state is only generated once. When, for example, during the first request a cookie is not present (because it has not

Client	Snort	Suricata	Bro
1	2 Denial of service alerts		
2	2 Denial of service alerts		
3	1 Denial of service alert		
4	2 Denial of service alerts		
5	2 Denial of service alerts		
6	6 Denial of service alerts		
Tool1	1 Denial of service alert		
Tool2	1 Denial of service alert		
Tool3	1 Denial of service alert		
Tool4	1 Denial of service alert		
Tool5	1 Denial of service alert		
Wget	1 Web-misc alert		Wget alert
	6 Web crawl alerts		
	12 Denial of service alerts		
	11 Web cgi alerts		

Table 5.5: The responses of the IDS on the sample set, `wget` and our created data for `http://nos.nl`

been received), while at a later point in time a cookie is received and the first request is sent again, then the last request does not contain the cookie. This is not realistic behavior

Design issues

Choices made in designing our tool can lead to abnormal behavior in terms of traffic. For example, the order of the requests are mixed. Although this is possible, it is rather unlikely to occur frequently. Another problem with this is that relatively complicated actions such as posting after authentication can cause problems. For example, a request is made to post information while not authenticated.

Differentiating from data also causes other problems such as that the think time is unrelated to the received response, while there is possibly a relation between the size of the response and the think time.

Also, using a few sample sessions to obtain the user states, as discussed in Section 4.2, results in a rather limited set of user states. Besides, if a user uses tabbed browsing and opens multiple tabs during a short time interval, then these requests are not detected as user states, while in fact they are.

The attempt to detect a language, as defined in Chapter 3.5.3 failed, since HTTP does not accept spaces in the token that defines the file requested, the information posted, etc.

Another problem, for which our tool has no support, is compressed responses. The compressed text appears to be random and no relations between requests and responses can be found.

5.3.3 Experimental data

Table 5.8, 5.9, 5.10, 5.11 provides information about the performance of our tool, while Table 5.7 provides information about the sample set.

Client	Snort	Suricata	Bro
1	8 Web-misc alerts		
	2 Denial of service alerts		
2	1 Denial of service alert		
3	3 Denial of service alerts		
4	10 Web-misc alerts		
	7 Web cgi alerts		
	4 Denial of service alerts		
5	4 Web cgi alerts		
	1 Web misc alert		
	1 Denial of service alert		
6	3 Denial of service alerts		
Tool1	1 Denial of service alert		
Tool2	1 Denial of service alert		
Tool3	1 Denial of service alert		
Tool4	1 Denial of service alert		
Tool5	2 Denial of service alerts		
Wget	5 Web crawl alerts		Wget alert
	10 Denial of service alerts		
	406 Web cgi alerts		

Table 5.6: The responses of the IDS on the sample set, `wget` and our created data for `http://tweakers.net`

The tables show some remarkable results. For example, Table 5.11 shows that our tool only makes 1 request. An analysis of the corresponding response reveals that the last 4 bytes are not a delimiter. The last character "\n" is missing. Therefore the response is not parsed correctly and a timeout occurs, ending the session.

Our tool also performs relatively bad for `http://nos.nl` and `http://www.voetbalzone.nl`. This is caused by malformed requests, where the requested file or "Host" specification is invalid.

Interesting is also to see that the number of theoretical requests is often significantly lower than those of the sample set. This is caused by wrongly defined user states. For example, when a request for an image is defined as a user state, then no protocol/client states are requested.

Another interesting point is that there is a discrepancy between the theoretical number user states that are sent, and the actual number of user states that are sent. This is caused by the fact that not all templates can be filled with the required information. If a state is not ready when it needs to be sent, it is simply ignored, causing the discrepancy.

Client	http://www.voetbalzone.nl	http://nos.nl	http://tweakers.net	http://www.security.nl
Client 1	167	138	226	38
Client 2	29	129	105	49
Client 3	210	82	61	74
Client 4	34	61	149	64
Client 5	88	168	142	76
Client 6	55	229	130	64

Table 5.7: The number of requests in our sample set for each client for each web site

Client	Requests in theory	Requests in practise	Malformed requests
Tool1	136	132	80
Tool2	27	19	7
Tool3	27	23	5
Tool4	46	39	15
Tool5	115	88	56

Table 5.8: Data for `http://www.voetbalzone.nl`

Client	Requests in theory	Requests in practise	Malformed requests
Tool1	114	90	66
Tool2	126	120	91
Tool3	105	100	92
Tool4	115	108	84
Tool5	178	135	89

Table 5.9: Data for `http://nos.nl`

Client	Requests in theory	Requests in practise	Malformed requests
Tool1	69	69	6
Tool2	90	88	5
Tool3	85	77	4
Tool4	123	69	8
Tool5	95	92	5

Table 5.10: Data for `http://tweakers.net`

Client	Requests in theory	Requests in practise	Malformed requests
Tool1	31	1	0
Tool2	28	1	0
Tool3	25	1	0
Tool4	29	1	0
Tool5	30	1	0

Table 5.11: Data for `http://www.security.nl`

Chapter 6

Conclusion

This chapter states the conclusion of our thesis. First the conclusion is given after which Section 6.1 discusses future work.

We started our thesis based on a relatively open description: “Data sets for IDSs”. First step was to explore the subject of IDSs, IDS testing principles, and input data for IDSs. This is discussed in Chapter 2. We saw that in the worst case scenario each type of IDS, for each testing purpose, for each environment it is in, needs its own input data for testing the IDS, in order to derive a proper conclusion. This chapter also discussed various types of input data for IDS such as data sets, frameworks and attack mutation tools. We saw that the focus of the scientific community has been largely on IDS evasion techniques, for which quite some advanced tools and information are available. This in contrast to methods that create benign artificial traffic, which is required by anomaly based NIDSs. Currently the method that comes the closest to benign artificial traffic for the application layer, is to mirror a web/ftp server using the `wget -m` command. The problem with using this as the only method to create benign traffic for IDS testing, is that it is easily detectable (for example a small think time between requests, the same client version is used, etc).

For those reasons, and because of the fact that information in today’s society is the most important business asset (see Chapter 1), we decided to develop a tool that solves the aforementioned problems. Our tool works in short as follows: A few sample sessions of an unknown plain text protocol are analyzed. Information is obtained and used to create a semi state machine. This semi state machine is able to interact with a server and thus create artificial traffic. By adding a mixing variance to the tool we are able to, based on the same sample sessions, create different semi state machines. This way it is possible to create traffic, based on a few sample sessions. The choice for having no prior protocol knowledge is useful for the case that the discussed techniques can be applied to other plain text protocols too, making traffic generation relatively easy. However, creating such a system is time consuming. Therefore we decided to focus on HTTP, which is the most used protocol, for which the most vulnerabilities are found.

The protocol analysis part is discussed in Chapter 3. First techniques to determine the delimiters are discussed. Once these have been found, they are used to tokenize protocol runs. In a protocol definition there can be variables and identifiers, where a variable is a value likely to be changed throughout different requests, while the identifier stays the same. These are important for the next part in that chapter, which discusses token types. A token can be defined by, for example, a user and the protocol/client. Besides tokens can also be dynamic, meaning that a variable has for example a relation with a token received in a response at an earlier point in time. For example, when a user visits a web page, the browser will request some images which are found in the response. The last part of this chapter discusses user profiles, which define the behavior of the user. For example, a user who uses the lynx web browser does not request CSS files.

The next chapter, Chapter 3 discusses the traffic generation part. The concept of a state is explained, which is basicly a template on how to construct a request based on the information from Chapter 3. The advantage of having a template is that the content of the request depends on the content of a response from that session, and not from a session in our sample set. The nex part in that chapter discusses several ways of obtaining statistics These statistics are later used to differentiate between various properties of a semi state machine. For example, the order of requests, the number of of requests by user, the think time, as well as the client profile can all be completely different based on the randomization. This way it is possible to create many different semi state machines with a limited sample set.

Chapter 5 discusses the testing of our tool. We obtained our sample set by having 6 different users visit 4 different websites through a proxy server, resulting in a sample set of size 6 for each website. Our tool is used with our sample set and for each website 5 different semi state machines are created. The next part is devoted to testing the quality of our tool. We had three IDSs analyze our sample set, the data created by our tool, and data created by `wget`. We saw that Snort, as well as Bro (with a special for this purpose crafted script) are able to detect the data created by `wget`. Our sample set as well as the data created by our tool did raise some alarms. However these alarms are false positives.

Unfortunately this does not mean that our tool creates realistic traffic. The first, and most important problem, are malformed requests. These are caused by the assumption that the content of a web page might change, but the layout does not change a lot. When the layout does change, the token that specifies the file can be malformed, resulting in a request for a file that is nowhere found in the response. The other, more general problem is the fact that traffic is mixed, since an IDS can be created/configured to detect this, making the traffic generation part of our thesis not entirely realistic. Also not all states can be sent, because for some the appropiate token cannot be found.

We conclude that we made a step forward into benign artificial traffic generation, that with our tool better benign artificial traffic generation is possible, and that the problems caused by using solely `wget -m` are overcome. The next section, Section 6.1 discusses the future work based upon this thesis. The future work is hopefully another step forward towards better benign artificial traffic generation.

6.1 Future work

The future work is divided into 3 parts. The first part, Section 6.1.1, discusses other application domains of the techniques proposed in this thesis. The second part, Section 6.1.2, contains general ideas for future work on traffic generation that apply for all types of protocols. The parts after this are protocol type specific and discuss often on which protocol parts statistics should be obtained, which again can be used to create traffic. Although not mentioned in each part, it is mandatory for each part to determine the statistical distribution. Also protocol classification is not always easy. Skype for example can be seen as a streaming-P2P-IM protocol.

6.1.1 Application domain

The mixing of data, as proposed in Chapter 4, can also be used for intrusion detection purposes. For example, client profiles from Section 3.6 can be combined with the states from Section 4.2 in order to build user profiles. Now however, differentiating as proposed in Section 4.4, does not need to be used since it has to be detected. Therefore request traces have to be made, with which typical user behavior can be simulated, and which can be used for anomaly detection. A request trace that differs too much from the known user traces is flagged as an anomaly.

In Chapter 5 a simple BRO script was written to detect `wget -m`. A flow based IDS can also detect this behavior. The in Section 4.2 proposed techniques to detect different types of states that can help with this. For example, when the think time between various user states is too small then traffic can be marked as malicious. The flow based IDS can also look at the size in terms of bytes of the responses to detect scripts that look for known vulnerabilities. For example when an attacker runs such a script, most, if not all requests, receive an HTTP 404 error. The sizes of the responses are usually around the same size, which can be detected by the IDS.

6.1.2 General future work for benign artificial traffic generation

The usefulness of the plain text protocol analysis part in Chapter 3 is debatable. Writing a protocol analysis analyzer instead of a protocol specific parser has its drawbacks. For example anomalies during a protocol run are undetected and can lead to a false analysis. Another problem is that some characteristics of a protocol are not detected. Because of these reasons, a specific protocol analysis part, based on the protocol definition, can be future work. Another advantage of this is that while creating traffic and interacting with a server, parsing can be faster, when the protocol is known in advance.

Today's protocols can include encryption, often being some form of asymmetric encryption in the beginning after which symmetric encryption takes over once a key is set. Future support for encryption functionality will greatly improve the quality of the to be produced data sets.

Text can still appear random when it is compressed, with the consequence that no relations between requests and responses can be found. Therefore compression algorithms need to be included too, so the protocol analysis part functions properly.

In this thesis we omitted some important aspects of how realistic the created traffic is when viewed from a higher level. For example, an important aspect is how to create the right quantity of traffic in terms of bandwith usage, or what is a realistic distribution of source IPs. Another part that is not researched are privacy issues with the sample sets. The sample sets can obtain sensitive data which can possibly be found in the created data. Also the way sample sets are obtained (by asking users to visit a website and setting a proxy server in their browser) is not very feasible. These interesting issues require further research.

6.1.3 IM protocols

For IM protocols it is important to create statistics on, for example:

- Think time
- Typing speed, since some IM protocols such as MSN Messenger and Skype tell the other party when the other party is typing.
- Language, word length, word frequency, etc. These should be defined user-channel/user. For example a user types different based on the chat room she is in, or the person she talks to.
- The duration and time a user is online.
- The frequency, size and type of files sent and received.
- The different client version and with it its associated behavior.

6.1.4 P2P file sharing protocols

P2P protocols can be relatively complicated, which is attributed to the fact that different client versions support different futures and act different. Creating a tool to create traffic is therefore a relatively hard task. General statistics to be obtained are, but are not limited to:

- The types of files requested, for example MP3, avi, etc.
- The search strings for a file and the relation between the search string and the to be downloaded file.
- Down- and upload caps.
- The different client versions and with it its associated behavior.

6.1.5 Request response protocols

Request response protocols such as NNTP and HTTP often have completely different requests and responses at different points in time, due to the retrieved information that is updated between the time interval. For this reason the obtained information should be parsed such that important information for the requests are not lost and used when necessary. For example in HTTP cookies are detected and used when necessary. General statistics to be obtained are, but are not excluded to:

- The types of files requested, combined with the client version.
- Think time

- Language, word length, word frequency, etc. Also if part of the request is found previously in a response. This is the case when a user replies to another user's message.
- The different client version and with it its associated behavior.
- The order in which requests are sent.

Of course in the case of HTTP, measures need to be taken to overcome the previous mentioned problems found in this thesis.

Appendix A

Bro wget detection script

```
const GET_REQ = "GET";
const WGET_MAX = 400;
const CLIENT_VERSION = "Wget";
event connection_established(c: connection)

{
    if ([c$id$orig_h] !in c_table){
        w_table[c$id$orig_h] = 0;
        c_table[c$id$orig_h] = 0;
    }
}

event http_request (c: connection, method: string, original_URI: string,
    unescaped_URI: string, version: string)
{
    if (GET_REQ in method){
        ++c_table[c$id$orig_h];

        if (c_table[c$id$orig_h] == WGET_MAX)
            alarm "WGET DETECTED BY COUNT";
    }
}

event http_header(c: connection, is_orig: bool, name: string, value: string)
{
    if (name == "USER-AGENT")
        if (CLIENT_VERSION in value)
            if (w_table[c$id$orig_h] == 0){
                alarm "WGET DETECTED BY VERSION";
                ++w_table[c$id$orig_h];
            }
}
```

Appendix B

Snort alerts

B.1 Denial of service alert

```
[**] [1:100000160:2] COMMUNITY SIP TCP/IP message flooding directed to SIP
proxy [**]
[Classification: Attempted Denial of Service] [Priority: 2]
05/26-22:02:09.412235 81.173.100.40:80 -> 10.109.0.1:54405
TCP TTL:122 TOS:0x0 ID:17971 IpLen:20 DgmLen:1405 DF
***A**** Seq: 0x52A12721  Ack: 0x4C4159AE  Win: 0xFE1C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 11259987 446071
```

B.2 Unicode alert

```
[**] [119:7:1] (http_inspect) IIS UNICODE CODEPOINT ENCODING [**]
[Priority: 3]
05/26-21:52:37.654144 10.107.0.1:47012 -> 213.156.1.80:80
TCP TTL:64 TOS:0x0 ID:27670 IpLen:20 DgmLen:632 DF
***AP*** Seq: 0x7EEDCE80  Ack: 0xFF673F3B  Win: 0x5C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 369260 1354125783
```

B.3 Web-misc alert

```
[**] [1:1852:3] WEB-MISC robots.txt access [**]
[Classification: access to a potentially vulnerable web application]
[Priority: 2]
05/30-00:43:50.831429 10.109.0.1:39768 -> 81.173.100.40:80
TCP TTL:64 TOS:0x0 ID:25603 IpLen:20 DgmLen:252 DF
***AP*** Seq: 0xD673B73  Ack: 0xA7132C86  Win: 0x5C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 3956596 0
[Xref => http://cgi.nessus.org/plugins/dump.php3?id=10302]
```

B.4 Web crawl alert

```
[**] [1:2002823:1] BLEEDING-EDGE POLICY POSSIBLE Web Crawl using Wget [**]
```

```
[Classification: Attempted Information Leak] [Priority: 2]
05/30-00:43:51.732269 10.109.0.1:39776 -> 81.173.100.40:80
TCP TTL:64 TOS:0x0 ID:56629 IpLen:20 DgmLen:320 DF
***AP*** Seq: 0xE9B82FE  Ack: 0x91E634A5  Win: 0x5C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 3956687 0
[Xref => http://www.gnu.org/software/wget/]
```

B.5 Web cgi alert

```
[**] [1:881:5] WEB-CGI archie access [**]
[Classification: Attempted Information Leak] [Priority: 2]
05/30-00:39:51.770716 10.108.0.1:38198 -> 145.58.28.91:80
TCP TTL:64 TOS:0x0 ID:13458 IpLen:20 DgmLen:293 DF
***AP*** Seq: 0x72C8873E  Ack: 0xDF68D096  Win: 0x5C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 3912621 506339454
```

B.6 Double decoding alert

```
[**] [119:2:1] (http_inspect) DOUBLE DECODING ATTACK [**]
[Priority: 3]
05/30-00:41:02.159548 10.108.0.1:38393 -> 145.58.28.91:80
TCP TTL:64 TOS:0x0 ID:2426 IpLen:20 DgmLen:401 DF
***AP*** Seq: 0xB4F9C654  Ack: 0x2239AEA8  Win: 0x5C  TcpLen: 32
TCP Options (3) => NOP NOP TS: 3919660 506346494
```

Appendix C

A new statistical distribution

A new distribution for creating statistics when the sample set size is limited is based on the following assumptions:

- The numbers that occur are more likely to occur since they occur.
- The minimum and maximum values found are used as boundaries.

The algorithm consists of 3 steps. First, all the numbers are sorted, after which a multiplication factor mf is determined. The multiplication factor specifies the sum of the differences between the sorted numbers[1]. Then a new array is created which will contain the new statistical data. Finally, the array is filled. All the numbers that have been found are presented mf times in the new array. All the number that were not found appear only 1 time in the array. This results in Algorithm 4:

With this algorithm the probability for a number found in the input is defined as $P_{new}(X)$ and the probability function based on the data as $P_{old}(X)$. Equation C.1 defines $P_{new}(X)$.

$$P_{new}(X) = \frac{P_{old}(X)}{src.size} \tag{C.1}$$

A problem with this algorithm is that for a large mf a large table is created.

[1]A difference is defined $number_{n+1} - number_n - 1$ where $number_{n+1} - number_n \geq 1$

```
**Input**: An array with data, *src*
**Output**: An array *ret* with statistics based on *src*
*sort_array(src)*;
$mf \leftarrow$ *find_multiplication_factor(src)*;
$ret \leftarrow new\_array(mf * src.size + mf)$;
**foreach** $i \leftarrow$ *offset* $\leftarrow 0$ **to** *ret.size - 1* **do**
    **foreach** $j \leftarrow 0$ **to** $mf$ **do**
        $ret[increment(offset)] \leftarrow src[i]$;
    **end**
    **if** $(diff \leftarrow src[i+1] - src[i] - 1) \geq 1$ **then**
        **foreach** $j \leftarrow 0$ **to** *diff* **do**
            *ret[increment(offset)]* $\leftarrow$ *ret[offset - 1] + 1*
        **end**
    **end**
    **foreach** $i \leftarrow 0$ **to** $mf$ **do**
        *ret[increment(offset)]* $\leftarrow$ *src[src.len - 1]*;
    **end**
**end**
```

Algorithm 4: `Create Statistics`

Bibliography

[1] Joel Sommers, Hyungsuk Kim, and Paul Barford. Harpoon: a flow-level traffic generator for router and network tests. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 392–392, New York, NY, USA, 2004. ACM.

[2] Joel Sommers, Vinod Yegneswaren, and Paul Barford. Toward comprehensive traffic generation for online ids evaluation. 2005.

[3] Joshua W. Haines, Lee M. Rossey, Richard P. Lippmann, and Robert K. Cunningham. Extending the darpa off-line intrusion detection evaluations. *DARPA Information Survivability Conference and Exposition,*, 1:0035, 2001.

[4] Lee M. Rossey, Robert K. Cunningham, David J. Fried, Jesse C. Rabek, Joshua W. Haines, and Marc A. Zissman. Lariat: Lincoln adaptable real-time information assurance testbed. In *In IEEE Proc. Aerospace Conference*, pages 2671–2682, 2002.

[5] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 82–87, New York, NY, USA, 2004. ACM.

[6] Ricardo Guttierez-Osuna. Introduction to pattern analysis. Lecture 7: Kernel Density Estimation.

[7] ISO. *ISO 17799: Information technology – Security techniques – Code of practice for information security management.* ISO, Geneva, Switzerland.

[8] Kim Zetter. `http://www.wired.com/threatlevel/2009/06/watt/`. Last accessed 17 April 2011.

[9] Kevin D. Mitnick and William L. Simon. *The Art of Intrusion: The Real Stories Behind the Exploits of Hackers, Intruders & Deceivers.* John Wiley && Sons, Inc., New York, NY, USA, 2005.

[10] William H. Allen. Mixing wheat with the chaff: Creating useful test data for ids evaluation. *IEEE Security and Privacy*, 5:65–67, 2007.

[11] Vidar Evenrud Seeberg. Anonymization of real data for ids benchmarking. Master's thesis, Gjovik University College, 2006.

[12] http://lcamtuf.coredump.cx/mobp/. Last accessed 2 April 2009.

[13] James P. Anderson. Computer security threat monitoring and surveillance. Technical Report Contract 79F26400, James P. Anderson Co., Box 42, Fort Washington, PA, 19034, USA, 1980.

[14] Rebecca Gurley. Bace and Peter. Mell. *Intrusion detection systems [electronic resource] / Rebecca Bace and Peter Mell.* U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, [Gaithersburg, MD :, 2001.

[15] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison-Wesley Professional, 2006.

[16] K. Scarfone and P. Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS).*

[17] Frederick B. Cohen. A short course on computer viruses. *Computers and Security*, 1990.

[18] Stefano Zanero. *Unsupervised Learning Algorithms for Intrusion Detection.* PhD thesis, Politecnico di Milano, 2006.

[19] Stefano Zanero. My ids is better than yours! or ... is it? Slides BlackHat-Briefings 2006.

[20] Song Luo. *Creating models of internet background traffic suitable for use in evaluating network intrusion detection systems.* PhD thesis, Orlando, FL, USA, 2005. Major Professor-Marin,, Gerald A.

[21] Michael Aaron Cooper. Honest: Hands-on networking environment for security testing. Master's thesis, Florida Institute of Technology, 2005.

[22] M. V. Mahoney and P. K. Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection*, September 2003.

[23] Pavel Laskov, Patrick Düssel, Christin Schäfer, and Konrad Rieck. Learning intrusion detection: Supervised or unsupervised? In *ICIAP*, pages 50–57, 2005.

[24] http://www.kdd.org/kddcup/index.php?section=1999&method=task. Last accessed 2 April 2009.

[25] http://cctf.shmoo.com/. Last accessed 2 April 2009.

[26] R. Bajcsy, T. Benzel, M. Bishop, B. Braden, C. Brodley, S. Fahmy, S. Floyd, W. Hardaker, A. Joseph, G. Kesidis, K. Levitt, B. Lindell, P. Liu, D. Miller, R. Mundy, C. Neuman, R. Ostrenga, V. Paxson, P. Porras, C. Rosenberg, J. D. Tygar, S. Sastry, D. Sterne, and S. F. Wu. Cyber defense technology networking and evaluation. *Commun. ACM*, 47(3):58–61, 2004.

[27] http://www.isi.edu/deter/tools.html. Last accessed 2 April 2009.

[28] `http://seer.isi.deterlab.net/trac`. Last accessed 2 April 2009.

[29] Michael Richmond. Vise: The virtual security testbed. Technical report, University of California, 2005.

[30] Gautam Singaraju, Lawrence Teo, and Yuliang Zheng. A testbed for quantitative assessment of intrusion detection systems using fuzzy logic. In *IWIA '04: Proceedings of the Second IEEE International Information Assurance Workshop (IWIA'04)*, page 79, Washington, DC, USA, 2004. IEEE Computer Society.

[31] `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`. Last accessed 2 April 2009.

[32] `http://erratasec.blogspot.com/2007/03/yet-more-blogging-blackhat.html`. Last accessed 2 April 2009.

[33] Davide Balzarotti. *Testing Network Intrusion Detection Systems.* PhD thesis, 2006.

[34] `http://www.metasploit.org`. Last accessed 2 April 2009.

[35] IBM X-FORCE. Ibm x-force 2010 mid-year trend and risk report. Technical report, IBM, 2010.

[36] Hendrik Schulze and Klaus Mochalski. Ipoque internet study 2008/2009. 2009.

[37] M. Rose. On the Design of Application Protocols. RFC 3117 (Informational), November 2001.

[38] `http://www.its.bldrdoc.gov/fs-1037/dir-011/_1544.htm`. Last accessed 2 April 2009.

[39] `http://www.google.com/uds/samples/language/detec.html`. Last accessed 1 December 2010.

[40] `http://www.alchemyapi.com/api/lang/`. Last accessed 1 December 2010.

[41] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[42] D. W. Scott. *Multivariate Density Estimation. Theory, Practice, and Visualization.* Wiley, 1992.

[43] `http://freehaven.net/~chrisd/polipo/polipo-1.0.4.1.tar.gz`. Last accessed 1 March 2010.

[44] `http://www.snort.org/downloads/917`. Last accessed 26 May 2011.

[45] `http://www.bleedingsnort.com/downloads/bleeding.rules.tar.gz`. Last accessed 26 May 2011.