Building IDS rules by means of a honeypot

Vidar Ajaxon Grønland

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2006

Institutt for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

In today's society people become more and more dependent on computer systems. It is therefore vital that such systems are up and running at all times. One factor that has the power to destroy the availability is computer network attacks (CNA). (CNA are defined as "methods aimed at destroying, altering or obstructing information in computers, computer networks or the networks themselves"). Unfortunately, the Internet show an increasing trend regarding the usage of malicious activities such as intrusion attempts, denial-of-service attacks, phishing, spamming and worms. Some automated attacks can compromise a large number of computers in a short period of time. To try to minimize this threat, it would be nice to have a security system which has the ability to detect new attacks and react on them. This thesis focuses on seeing how good IDS rules that can be generated automatically based on data logged by a simple honypot. The results are based on data collected by a network intrusion detection system named `SNORT`, a low-interaction honeypot named `honeyd` and a vulnerability scanner named `Nessus`. Results found in this thesis claim that `honeyd` logs sufficient information to make functional `SNORT` rules out of, but some rule options are not possible to determine.

Sammendrag

Dagens samfunn har blitt veldig avhengig av datasystemer for å være produktive. Det er derfor viktig at disse systemene har en høy grad av tilgjengelighet (oppetid). En faktor som har muligheten til å redusere tilgjengeligheten er datanettverks angrep (CNA). (CNA defineres som metoder hvis hensikt er å ødelegge, endre eller hindre informasjon i datamaskiner, datanettverk eller nettverket i seg selv). Dessverre er det en økning i antall angrep over internett fra år til år. Disse angrepene inkluderer blant annet innbrudd, DoS, phishing, spam og ormer. Noen av disse angrepene kan "ødelegge" et stort antall maskiner på kort tid. For å redusere denne trusselen er det ønskelig å ha et system som har muligheten til å se nye angrep og behandle disse. Denne masteroppgaven vil fokusere på hvor gode IDS regler som kan bli generert automatisk basert på informasjon hentet fra en enkel honeypot. Resultatene er basert på data samlet inn av en nettverks IDS ved navn `SNORT`, en lav interaksjons honeypot ved navn `honeyd` og en sårbarhets skanner ved navn `Nessus`. Resultatene våres viser at `honeyd` logger tilstrekkelig informasjon til å kunne generere funksjonable `SNORT` regler, men enkelte felter i reglene er ikke mulig å finne basert på loggene.

Acknowledgements

I have been fortunate enough to receive help and support in all phases of this thesis. I am especially grateful to Professor Slobodan Petrović who has provided guidance, expertise and encouragement during the thesis work. My fellow students also deserve thanks for giving me inputs during master seminars and discussions. I would also like to thank members of the SNORT forum, and the honeyd forum for giving answers to my questions. I also want to thank my family and friends for the patience and understanding they have shown for my work. Last but not least, my thanks go to my girlfriend for her love, support, and understanding during the whole period of time that went into this thesis.

Vidar Ajaxon Grønland, 2006/06/30

Contents

List of Figures

List of Tables

1 Introduction

1.1 Topic covered by this thesis

This thesis covers issues regarding behaviour and implementation of a simple honeypot and the use of such technology in creating IDS rules. A honeypot is a computer that is implemented in a network for the purpose of attracting attackers. This computer has nothing to do with the production network, thus all traffic into the honeypot is by definition malicious [2]. The goal is to attract attackers by pretending to be an interesting network. The log files from the honeypot serve as data collectors in conjunction with other widely used data collectors such as `tcpdump` [23] if needed. A security scanner named Nessus [4] is used to generate traffic towards the honeypot, leaving us with full control of the entire system. We use `SNORT` [8], a signature based Network intrusion detection system (NIDS) to check if the rules we create are usable. The main goal is to see how good `SNORT` rules that can be made, with as little user intervention as possible, based on information from the collected data.

1.2 Problem description

Many IDS's in use today are signature based. These IDS' are only capable of detecting already known attacks (attacks which have a signature entry in the database of the IDS). This is a huge problem when new attacks arrive. A signature based IDS are only capable of detecting alterations of already known attacks at best. Therefore there is an interest in trying to make a rule generating system to automatically generate new rules when new attacks arrive. In this thesis we look at the possibility of using a low-interaction honeypot to address this problem. The important question is then if the honeypot logs sufficient information to make rules out of. We propose a measurement method to see how good the rules we create are, compared to original rules alerting on the same threat. Due to the limited time available, we only re-create known rules and measure the difference between the originals and those we re-create.

1.3 Justification, motivation and benefits

There are issues concerning the effectiveness of IDS' capability to alert on malicious traffic. By using honeypots, one can be certain that all traffic towards it is malicious. This is because the honeypots itself have no production activity and no authorized activities. The question is then if the IDS' are capable of detecting all the attacks in the honeypot, or if some attacks are undetected. By checking the logs from the honeypot, it is possible to check if the IDS detected all the attacks, and if it did not, there have to be made new rules. Measuring how good rules we are able to create is an important question that has to be answered.

Signature based IDS vs. Anomly based IDS

As stated earlier, signature based IDS' are good at alerting on already known threats. They usually tell the administrator what type of attack it has detected (portscan, web,

ftp etc.), and their importance (priority). It is of course important that the IDS rule is of good quality to be sure the alerts are trustworthy. The problem with signature based IDS' occur when a new attack arrives. In these cases, anomaly detection is supposed to function well. In anomaly detection we know or learn the nature of the normal, non-attacked state. Then we compare the present state with what is supposed to be normal. The system then alerts if it sees behaviour, which is different from the normal one. The challenge to anomaly-based detection is defining what is normal. Since most networks change over time (adding services etc.), the "normal" state also changes, making it difficult to establish the normal network behaviour for longer periods. This is the reason for us to choose a signature based IDS in this thesis. We want to make it easier to create new rules for new attacks to be used in a signature based IDS. One solution is to actually use an anomaly based IDS to direct possible malicious traffic to the honeypot as suggested by Anagnostakis et al. in [13], and then make signatures if the traffic is deemed malicious.

Benefits from using a honeypot when creating new IDS rules

Honeypots are a security technology whose value is in being attacked, probed or compromised so that the methods of the attacker can be logged and studied. The philosophy is simple: they do not have any production purpose, there is no authorized interaction with them, so any interaction with a honeypot is most likely a scan, probe or attack. The reason for choosing honeypots in this thesis is because of their detection value. As stated, a honeypot has no production activity, no authorized, legitimate interactions will take place on it. This basically means that all traffic towards it is by definition malicious, leaving us with a dataset only consisting of malicious activity. In a way, one may see a honeypot as the opposite of an IDS. Where IDS' fail, honeypots can excel. Here are several examples:

- Reduces False Positives - All activity with the honeypot is by definition unauthorized, making it extremely effective at detecting attacks.
- Detecting False Negatives - A honeypot easily identifies and captures new attacks or activities against it.
- Encryption - Even if an attack is encrypted, the honeypot will still capture the activity.

Honeypots can carry out extremely simple and cost effective detection. They usually need very little resources and maintenance, leading to a very cheap implementation cost. Also the fact that the amount of information needed to be collected and analyzed is greatly reduced, allows employees to focus on other matters. This makes a honeypot perfectly suited for this research, both because of the data collection and the low implementation cost. We will use a simple honeypot to see if it is suited for generating rules to be used by a signature based IDS'. The reason for choosing a simple low-interaction honeypot is because of security aspects involved when using high-interaction ones.

1.4 Research questions

The following research questions have been answered in this thesis:

1. Is `honeyd` suitable for detecting new attacks?
2. How to translate data captured by `honeyd` into `SNORT` rules?

3. How good rules can be made based on data logged by `honeyd`?

1.5 Limitations

Because of the fact that the honeypot might log vast amounts of data for manual analysis, we have to limit ourselves in order to adjust the size of the task to the available resources. The procedure(s) proposed in this thesis are not a final product ready for production, but rather a prototype showing a proposed way of solving the problem. We have also limited the thesis to only include attacks against FTP port 21 and WEB port 80.

2 Previous work

2.1 What is an intrusion?

As far back as in 1980, the concept of intrusion detection was introduced. Anderson [14] defined an intrusion attempt or a threat to be the potential possibility of a deliberate unauthorized attempt to:

- access information (Confidentiality)
- manipulate information (Integrity)
- render a system unreliable or unusable (Availability)

This is still the case today, 26 years later. In [17], Biermann et al. define a computer intrusion to be any set of actions that attempt to compromise the Confidentiality, Integrity or Availability (CIA) of a resource, which is the same as stated by Anderson. Mukherjee et al. define intrusions as "unauthorized use, misuse and abuse of computer systems [30]. These definitions all lead to the same general observation, namely that an intrusion is a successful violation of the security policy.

RFC2828 [12] defines a security intrusion as:

> A security event, or a combination of multiple security events, that constitutes a security incident in which an intruder gains, or attempts to gain, access to a system (or system resource) without having authorization to do so.

2.2 Honeypot technologies

When deciding upon which honeypot type that was best suited for this master thesis, different types of honeypots were studied. The honeynet project [1] and the book [40] provided guidelines on how to implement honeypots. A program called Honeywall, which includes logging software (SEBEK [7]) and intrusion prevention system (SNORT-inline) is described in [1]. SEBEK is a tool for collecting data passing through the `read()` system call. This is useful for example when logging SSH connections, since this tool then logs all keystrokes and files used with the `scp` command. SNORT-inline is an intrusion prevention system which drops packets that would raise alerts on `SNORT`. Honeywall works as a bridge which is transparent to the attacker.

Spitzner [41] claims that for detection, simple honeypots that emulate systems and services, such as `Specter` [9] and `honeyd` [34], are the best. `Honeyd` [34] is a low interaction honeypot, able to simulate big networks with many services. For more information about `honeyd` please see section 5.2.

Another article related to honeypots is [33], which states: "complementarities between high and low interaction honeypots can increase the accuracy of information collected by simple environments deployed in different places." This means that one may use a high interaction honeypot to control what services need to be refined in order to

collect the same amount of data on the low interaction honeypot. This is usually solved by implementing scripts on the low interaction honeypot to simulate a response to a request from the cracker[1], as `honeyd` [34] does.

In [18], it is discussed how to attract attackers to the honeypot. Registering the honeypot in the DNS servers of the organisation and enabling zone transfer in the local DNS server of the honeypot is suggested. This approach might be useful if the honeypot we deploy has a low frequency of attacks.

Anagnostakis et al. [13] presents a system to combine the advantages of anomaly detection and honeypots. By using high-interaction honeypots, the amount of false positives seen by an anomaly detector can be drastically reduced due to the fact that all traffic towards the honeypot is deemed malicious by definition. The authors propose to mirror suspicious traffic to a honeypot previously detected by a network based anomaly detector to determine the accuracy of the anomaly prediction. Misclassified benign traffic will be validated by the shadow honeypot and then handled by the real system. They have named the system "shadow honeypot" because the traffic is transparently mirrored to the honeypot. The shadow honeypot is an instance of the protected application (e.g. a web server/client, FTP server/client) which shares all internal state with a normal instance of the application. This instance is instrumented to detect potential attacks. It is stated that shadow honeypots allow detecting exploits for client programs such as web browsers that require user interaction. This approach is in essence a way to train the anomaly detectors, but is also a possible approach we could use if we have sufficient time to implement a high-interaction honeypot. By using this approach it is possible to redirect possible malicious traffic to the honeypot, hopefully leaving us with more data to create rules from.

In order to find out how to translate honeypot captured data into a new signature, a study on how a `SNORT` [8] signature is built is obviously needed.

Problems with honeypots are well known. In [43], Spitzner discusses the problems with today's honeypots. Especially high-interaction honeypots have problems one should not oversee. These systems run real operating systems and applications and hence a serious security risk submerges. It is very important to think through all aspects of implementing such technology. Also the aspects of storing information about attackers are much debated. It is important to make sure that the laws of the country the honeypot is deployed in are followed. (In Norway, personopplysningsloven [5]). The main reason for us not to use a high-interaction honeypot is because it should be monitored (by a person) at all times.

In [36], a powerful tool for testing NIDS is described (AGENT [2]). This tool generates different ways of deploying a single attack. By doing this, one can be sure that the NIDS is capable of detecting variations of one specific attack. This tool might come in handy if

[1] Malicious hackers are often called `black hat hackers`, but it is more appropriate to call them `crackers` as this is a term which distinguishes the exploitation of security weaknesses from hacking in general.

[2] Attack GEneration for Nids Testing tool

we are not able to collect enough data in the honeypot, and it if is possible to direct the tests towards the honeypot instead of the NIDS.

2.3 Rule generating systems

The book "Intrusion Signatures and Analysis" [21] gives a good understanding about intrusion analysis methodology. It focuses on full analysis of an attack along with traces to determine what happened and how.

There have been a few attempts to design systems with self generating signatures based on logging information. One of the first attempts was Honeycomb [26]. Honeycomb is a host-based intrusion detection system that automatically creates signatures. It uses a honeypot to collect malicious traffic, and applies the longest common substring (LCS[3]) algorithm on the packet content of a number of connections going to the same services. Honeycomb applies the LCS algorithm to binary strings built out of the exchanged messages in two different ways: Horizontal detection [4] and Vertical detection[5]. The result from the LCS algorithm is then used as a signature when creating a new rule for the respective attack. An evaluation of Honeycomb performed in [49], states that Honeycomb produces too many signatures on traffic without malicious content. This results in a high number of false positives.

The article [49] also evaluates three other signature generation systems named Nemean [49], Earlybird [39] and Autograph [24]. The last two systems only make rules for worms, while Nemean generates rules in general.

Autograph [24] uses heuristics to classify traffic into two categories: one flow pool with suspicious activity and one with non-suspicious activity. To become a suspicious connection, more than S unsuccessful connection attempts towards different internal IP's must occur (all from the same source). Then the successful connections from this suspicious source are stored in the suspicious flow pool. A TCP flow reassembly is used on the suspicious flow pool, before they use Rabin fingerprints [19] to partition the payload into small blocks. Then the blocks are counted to determine their prevalence, and the most common substring from these blocks makes the worm signature. A blacklisting system is used to decrease the number of false positives.

Earlybird, the signature generating system proposed in [39], monitors traffic using sensors *sifting* through the traffic aiming at creating signatures for worms. The sensors report anomalous signatures to an aggregator responsible for activating blocking services, reporting and control. They propose an unfeasible algorithm, and end up using approximations to this algorithm. Their approach is based on two observations:

- Some portions of the content in worms are invariant

[3]The Longest Common Substring problem looks for the longest shared byte sequences across pairs of connections.

[4]Horizontal pattern detection: two messages at the same depth into the stream are passed as input to the LCS algorithm for detection.

[5]Vertical pattern detection: for both connections, several incoming messages are concatenated into one string and then passed as input to the LCS algorithm for detection.

- The spreading of a worm is atypical for internet applications

Said in a simpler manner, they claim it is rare to observe the same string recurring within packets sent from many sources to many destinations. The main procedure they propose is to *sift* through network traffic for content strings that are both frequently repeated and widely dispersed. This is supposed to be enough to identify new worms and their signatures. All false positives the authors of [39] have experienced have been feasible to "whitelist".

Nemean [49] incorporates protocol semantics into the rule generation algorithm. Because of this it is capable of handling more than just worm attacks. In comparison with COVERS [27], Nemean does not have a correlation step to pinpoint attack-containing bytes. COVERS only needs simplified message format specifications, while Nemean requires more detailed knowledge of service semantics. The biggest difference though is that COVERS do not require expert knowledge about which message fields are most likely to contain attacks as Nemean does.

Liang and Sekar propose an approach they call COVERS (COntext-based VulnERability-oriented Signature) [27], which uses a forensics analysis of a victim server's memory to correlate attacks to inputs received over the network, and automatically develop a signature that characterizes inputs that carry attacks. In short, they observe ongoing attacks -> create signatures -> use these signatures to filter out future occurrences of these attacks. The filters can be deployed as an in-line network filter or inside the address space of a protected server. The main advantage by the approach presented here is that it is able to generate signatures from single attack instances, as opposed to Earlybird, Autograph, Honeycomb, Polygraph [31] and PADS [45], which need sufficient number of attack samples to extract a good signature.

The Honeyanalyzer [46] is another signature generating system which generates rules based on `honeyd` captured information in conjunction with `tcpdump`. The security administrator (SA) gets information from a web GUI that tells him for instance what ports have been attacked. It is then up to the SA to execute the LCS algorithm on the log information that corresponds to the attack. The experience and wisdom of the SA is important to generate good signatures. This system has a high user interaction factor.

The author of [31] states that systems such as Honeycomb, Autograph and EarlyBird have a flaw in their assumption: "*that there exists a single payload substring that will remain invariant across worm connections, and will be sufficiently unique to the worm that it can be used as a signature without causing false positives.*" He states further that worm authors may design worms, which substantially change their payloads for each connection, so called polymorphic worms, which render the above mentioned systems useless. The contribution in [31] aims at producing signatures that match polymorphic worms. The major problem is that the payload of this type of worm varies as stated above. Therefore, signatures that are specified by a single contiguous non-variant sequence of bytes (one static signature used for content matching) can in general not be applied to detect these worms. Instead, the authors propose signatures that are specified by sets of contiguous byte sequences (Tokens). They present Polygraph, a system to cope with the above men-

tioned problems. The Polygraph monitor consists of a Flow classifier, a Polygraph Signature Generator, and a Signature Evaluator. The Flow classifier reassembles the payload of TCP connection and detects suspicious flows that are passed to the Signature Generator. In this article the authors concentrate on the Signature Generator and omit the design of the Flow classifier since this is well documented by i.e. [32]. The authors propose three types of signatures produced by the Polygraph Signature Generator:

- Conjunction Signatures
- Token-Subsequence Signatures
- Bayes Signatures

In [47] a system for creating rules for worms is presented. They claim that a collaborative security system (a distributed detection system that automatically shares information in real-time about anomalous behaviour experienced at the moment of attack among collaborating sites) will substantially improve protection against wide-scale infections. They use PAYL [48], an anomalous payload sensor, in a collaborative security system, and exchange information about suspected malicious packets. PAYL uses either LCS or LCSeq[6] on packets targeting the entire LAN, not only the honeypot. Since most worms have a self replicating behaviour, the authors assume that portions of incoming and outgoing malicious traffic are correlated that are sent by self replicating worms. If a correlation in the traffic is detected, the resulting data serves as a signature candidate. It is stated that their approach can detect zero-day exploits and generate signatures for these new exploits.

In [45] Tang et al. introduce an approach to automatically capture spreading worms. They introduce signatures ("position-aware distribution signatures (PADS)") that are based on a statistical measure to detect polymorphic worms. They use two honeypot types called inbound and outbound to capture worms. The inbound honeypot is a high-interaction intended to be compromised. This makes all traffic originating from this honeypot as being malicious. Then this traffic is sent to the outbound honeypot for analysis and signature generation. This system of two honeypots is denoted as a double-honeypot system.

[6] Longest Common Subsequence

3 Summary of claimed contributions

This research will contribute in the following ways:

1. See if a low-interaction honeypot (`honeyd`) is a good tool for logging attacks.
2. Ease the workload of the administrator when new IDS rules have to made. Signatures are to be used with `SNORT`.
3. See if data collected by the honeypot is sufficient for creating new IDS rules, and how good these rules can be made with as little user intervention as possible.

If we are able to determine what honeypot type collects the best data set, it will be easier for the others to perform similar experiments. It is not necessary for them to waste time on conducting experiments to determine what honeypot type they should use. It is vital though that the data set contains all the necessary log items that distinguish a specific attack from other attacks.

By reducing the user intervention regarding signature generation, the workload for the security administrator is reduced when it comes to keeping the IDS up to date. It will be much easier for the security administrator to find out that a new attack has occurred, and how he can make a signature to prevent the attack from happening again. This will improve the security of the system since the probability that a new attack goes through undetected is reduced.

We also present a measuring system for measuring how good the rules we manage to create from the honeypot logs are. This measuring system is described in 6.4.

4 Choice of Methods

We will use a mixed approach in this thesis, using literature studies and laboratory experiments. The choice of methods is described related to each research question.

1. Is honeyd suitable for detecting new attacks?
2. How to translate data captured by honeyd into SNORT rules?
3. How good rules can be made based on data logged by honeyd?

Is honeyd suitable for detecting new attacks?

Literature study and implementing honeyd to see how good it is.

How to translate data captured by honeyd into a new rule?

How to translate honeypot captured data into a SNORT rule will involve both a literature study of SNORT and a trial and error experiment to see if it is possible to get useful data from the honeypot logs.

How good rules can be made based on data logged by honeyd?

A method for measuring differences between original rules and rules we re-create will be presented. This will in essence measure if the new rules are missing important rule header or rule option fields. Also the false positive and false negative rate will be measured.

5 Theoretical background information and introduction to the new rule generating approach

5.1 SNORT

SNORT [8] is a widely used signature-based Intrusion Detection System. We first give a general definition of an IDS, and then explain characteristics of SNORT, as an important representative of such systems. The information below is given to show all elements, which must be considered when making a rule generating system.

Intrusion detection is a set of techniques and methods used to detect suspicious activity both at the network and host level. There are two basic categories of IDS', signature-based intrusion detection systems and anomaly detection systems.
Attacks have signatures (like computer viruses), that can be detected using software. Based upon a set of signatures and rules, the detection system is able to find and log suspicious activity and generate alerts. Anomaly detection is based on the assumption that an attack on a computer system will be noticeably different from normal system activity, and an intruder will exhibit a pattern of behaviour different from that of the normal use [44].

Intrusion detection systems usually capture data from the network and apply their rules to that data or detect anomalies in them. SNORT is primarily a rule-based IDS, and is our choice of IDS in this thesis. The reason for us to choose SNORT is because it is one of the most used open source[1] IDS'. SNORT uses rules stored in simple text files that can be modified by a text editor, which makes it easy for us to manipulate them. SNORT stores rules in separate files based on the category of the rule (ftp, web, snmp, icmp etc.). Technically, the rules are included in the main configuration file (`snort.conf`). SNORT reads this file at start-up, and builds an internal data structure as described in section 5.1.3. One important thing about SNORT is to implement as many signatures as we can using as few rules as possible. This is because SNORT gets slower for each added rule as described in section 5.1.3.

SNORT consists of several components as shown in figure 1. The major components are:

- Packet decoder
- Preprocessors
- Detection engine
- Logging and Alerting system
- Output modules

[1] Open source is a term used on software which is free of charge, and the source code is available for reading/editing

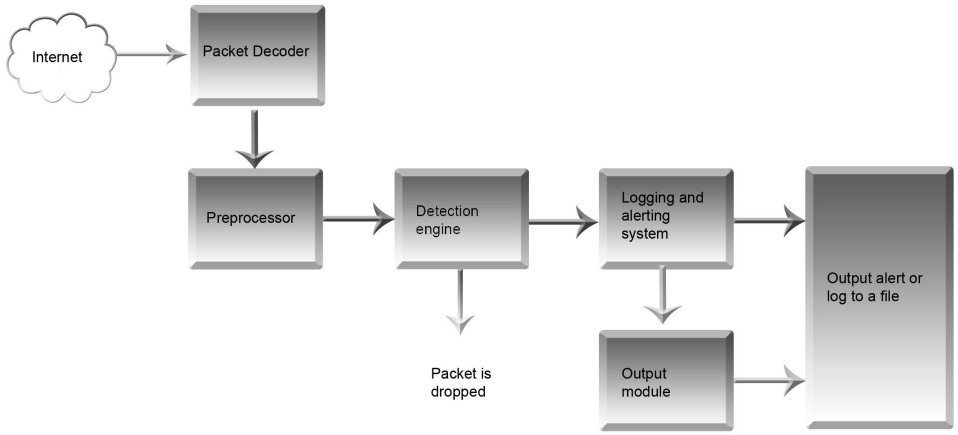

Figure 1: SNORT components

5.1.1 Packet decoder

The packet decoder is a series of decoders that each decode specific protocol elements into an internal data structure. It starts with the lower level Data Link protocols, and works it way up the network stack decoding each protocol as it moves up. When packets move through the decoders, a data structure is filled up with decoded packet data. Now the data stored in the data structure is ready to be analyzed by the pre-processors and the detection engine. Libpcap is used to capture the raw packets, this makes sure all protocol headers are unaltered by the OS.

5.1.2 Pre-processors

There are two categories of pre-processors. One purpose is to make the packet suitable for the detection engine to apply rules to it. The other purpose is to find obvious errors and detect anomalies in the data packets. Some attacks cannot be detected by signature matching using the detection engine. Because of this, special pre-processors have been made to try to detect these attacks. These pre-processors are vital in discovering non-signature based attacks. The other pre-processors tasks are to normalize traffic in order for the detection engine to accurately match signatures. The main goal for these pre-processors is to defeat attacks that try to evade the detection engine by manipulating patterns in the traffic. Defragmenting packets is also a task for the pre-processor. This is vital because before any rules may be applied, the packet must be reassembled. The reason for this is to avoid being misled by attacks that have been divided into several packets. pre-processors used by `SNORT` in default configuration are listed below.

- frag3 => Eliminate IP fragmentation attacks. Should always be enabled.
- stream4 => Used for maintaining the state of TCP streams, which is used in detecting some types of information gathering attacks. Important because signatures may be distributed amongst several packets.
- flow => Keeps track of TCP states.
- sfPortscan => Designed for detecting the first phase in a network attack: the Recon-

naissance. Was designed for detecting most NMAP[2] scans.

- HTTP_inspect => Detects abnormal HTTP traffic and normalizes it so that the detection engine can properly interpret it. The normalizing process translates various character sets, such as Unicode or hex, to characters that SNORT recognizes. It works specifically with the URI strings of an HTTP request. If the traffic encountered needs decoding an alert is generated. Works on a packet to packet basis (regardless of the fragmentation) unless another pre-processor reassembles the packets first.
- RPC_decode => Works in the same way as the HTTP_decode, but whith the RPC protocoll. Useful to avoid RPC attacks where the signature has been split into several packets.
- BO => Detects Back Orifice attacks. Specific for Windows systems.
- Telnet_decode => Decodes or removes arbitrarily inserted binary Telnet control codes in a Telnet or FTP stream. This eliminates the possibility to insert control codes into FTP or Telnet communications to avoid beeing seen by SNORT.
- Performance Monitor => Measures SNORT's realtime performance.
- ARPspoof => Detects some ARP spoofing attakcs like ARP cache overwrite and ARP spoofing.
- ASN1_decode => Detects various inconsitencies in ASN.1[3] which may indicate malicious behaviour.
- spade => Statisical Packet Anomaly Detection Engine. Used to detect general packet anomalies in IP packets. Uses a lot of memory on high load networks.

5.1.3 Detection engine

The most important part of SNORT is the detection engine. It serves two major functions: parsing rules and detecting signatures (intrusion activity). By parsing the SNORT rules, the detection engine builds attack signatures. The rules are read line by line, and load into an internal data structure (important to write the rules correctly, or the detection engine will fail when loading them into the internal data structure). Now all traffic is run through the loaded rule set in the order they were loaded into memory. Rules are split into two functional sections: rule header and rule option. In the rule header, information about conditions for applying the signature is set. This is the part in the rule before the parenthesis as shown in section 5.1.6.

The detection engine is the time-critical part of SNORT. Depending upon how powerful the host computer is and how many rules have been defined, it may take different amounts of time to respond to different packets. If the traffic on the network is too high, SNORT may start dropping packets if it does not have available resources to perform the signature matching. To be able to run SNORT on a high bandwidth network (above 100Mb/s), either a distributed system or a host with high specs are needed. This is a

[2]Nmap ("Network Mapper") is a free open source utility for network exploration or security auditing. It was designed to rapidly scan large networks, although it works fine against single hosts.

[3]Abstract Syntax Notation One is an international standard for coding and transmitting complex data structures. It is used by several higher-level protocols including LDAP, SNMP, SSL and X.509.

factor we must have in mind when designing the rule generating system. It is crucial to keep the number of rules down to a minimum in order to avoid dropping of packets, and make the rules as effective as possible.

The detection engine processes rule headers and rule options differently. A linked list decision tree is built by the detection engine. A packet is tested to see whether it is TCP. If so, the packet is passed to the part of the tree that has rules for TCP. Then the packet is tested to see if it matches a source address in a rule. If it does, the packet is passed down the corresponding rule chains. This procedure is done until the packet matches a signature, or tests clean. When a signature is matched to the content of a packet, SNORT makes an alert and continues searching for other rules matching the signature. On earlier versions of SNORT (1.x) this was not the case. SNORT used to stop checking when a signature match was found. This made it important to sort the rules in an order based on the most malicious signatures first. Because of the way SNORT handles rules, it is important to make sure that all necessary information is included in the rule to make it traverse the decision tree the right way. If this is not taken care of, then the rule might end up in the wrong place in the tree, leading to a false negative.

5.1.4 Logging and alerting system

Depending upon what the detection engine finds inside a packet, the packet may be used to log the activity or generate an alert. Our implementation of SNORT logs in simple text files and tcpdump-style files. All of the log files are stored under /var/log/snort folder on a UNIX system by default.

5.1.5 Output module

These modules are used to control the output from SNORT detection engine. Normally the alerts and logs go into files in the */var/log/snort* directory. By using these output modules, outputs can be processed and messages can be sent to a number of different destinations. Here are a few examples of output modules:

- Database
- SNMP, send alerts as SNMP traps to a centrally managed network operation center
- SMB, pop-up alert windows
- Syslog, for logging to a centralized logging server
- XML
- CSV, for comma separated files

5.1.6 Writing good SNORT rules

The purpose of a rule is to detect a specific type of traffic by matching all traffic against it. This is not always easy to accomplish. Often there is a gap between what the rule is intended to trigger on and what type of traffic actually triggers the rule. When writing the rules, one must narrow down the rule to only trigger on the traffic patterns of which alerts are wanted. This is really a balancing act, since a too specific rule most likely will fail to alert on attacks which deviate a little from the properties in the rule (false negatives). On the other hand, a rule which is too general will produce alerts on benign traffic (false positives). To be able to write good rules, it is important to research and

find unique patterns in the traffic. The patterns need not be unique by themselves, but if one combines them they should add up to be as unique as possible. Regarding the syntax of the rule it is vital to be sure that all elements in the rule are correct. If a faulty written rule manages to load, it could possibly trigger on large amounts of benign traffic, which in turn may lead to overload of the intrusion database. Another possibility is that the rule might not trigger on the intended traffic, leading to the belief that no attacks have occurred. Because of all possible problems with faulty written rules, it is suggested that all new rules are tested before they are implemented in production networks [25].

Rule header and rule option

A SNORT rule is divided into two parts, the rule header and the rule option. All text before the first parenthesis is the rule header. All text inside the parenthesis is the rule option. Let's take a closer look at the rule header. This header consists of the following:

- Rule action
  - Alert - Generate an alert and log the packet
  - Log - Just log the packet
  - Pass - Drop any packet that matches the signature
- Protocol - Monitor either TCP, UDP or ICMP packets
- Direction operator - What traffic direction the rule shall apply to
- Source and destination IP address
- Source and destination ports

The rule option consists of this:

- Content related options keyword
  - Content - Specifies the pattern to look for in the packet's payload
  - Uricontent - Triggers on the URI portion of a request
  - Nocase - Match regardles case
  - Offset - Where in the payload SNORT should start looking for the string specifeid in *content*
  - Depth - How many bits SNORT should look into the payload. Saves resources
  - Regex - Makes it possible to use regular expressions in the *content* part
- Session related option keyword
  - Flow - Specifies the traffic flow direction in which the rule should apply
  - Session - Captures and records session data
- IP-related option keywords
  - Ttl - Search for packets with this exact ttl value (IP header)
  - Tos - Search for packets with this tos value (IP header)

  - Id - Test the packet for a specific fragmentation ID
  - Ipopts - Monitors packets for specific IP options
  - Fragbits - Checks the fragmentation field in the IP header
  - Dsize - Detects a packets payload size
  - Ip_proto - Specifies what IP protocol to apply the rule to
  - Sameip - Checks if the source IP is the same as the destination IP
  - Fragoffset - Monitors for packets containing a particular fragmentation offset value
- TCP-related option keywords
  - Flags - The flags keyword is used to check if specific TCP flag bits are present
  - Seq - The seq keyword is used to check for a specific TCP sequence number
  - Ack - The ack keyword is used to check for a specific TCP acknowledge number
- ICMP-related option keywords
  - Itype - Is used to check for a specific ICMP type value
  - Icode - Is used to check for a specific ICMP code value
  - Icmp_id - Is used to check for a specific ICMP ID value
  - Icmp_seq - Is used to check for a specific ICMP sequence value

  Here is an example of a SNORT rule:

  alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg: "mountd access";)

This rule describes an alert that is generated when SNORT matches a network packet with all of the following attributes:

- *TCP* packet
- Source from *any* IP address on *any* port
- Destined for *any* IP address on the *192.168.1.0/24* network on port *111*
- Packet contains 00 01 86 a5

The word(s) before the colons in the rule options section are called option keywords. These keywords may appear once, as with *content* in the example above, or multiple times. If the rule above matches a packet, an alert is made with the message: mountd access. This makes it easy to figure out what the attacker was trying to accomplish.
If it is necessary to log both parts in a connection, one can use the bi-directional operator <>. This is handy for logging and analyzing both sides for instance in a telnet connection. The operator considers both IP address and port number. The following example is of a telnet connection logger:

log !192.168.1.0/24 any <> 192.168.1.0/24 23

This rule tells SNORT to log all packets between machines which is not on the 192.168.1.0/24 segment and machines on the 192.168.1.0/24 segment, where the destination port number is 23.

Recomended options to include when writing SNORT rules:

- Use the *msg* keyword
- Use the *classtype* keyword (or *priority* directly)
- Use a number to identify a rule by using the *sid* keyword
- If the rule applies to a known wulnerability, a reference to a URL should be given in order to make it easy to find more information about the attack. This is done by using the *reference* keyword
- Use the *rev* keyword to keep track of different verisons of the rule
- Use *flow* to make SNORT perform faster

It is important to make sure the signature/rule is specific to the service it applies to. For instance a rule triggering on the content: 4773903ac4b83ff4dc2s, must be specific to which service this string is a threat. If this is not done appropriately, a simple e-mail containing the string would generate an alert.

5.1.7 Understanding Standard SNORT Alert Output

SNORT writes alerts to the file */var/log/snort/alert* by default (Unix). When SNORT generates an alert message for a XMAS scan, it will look like this:

```
[**] [1:1228:7] SCAN nmap XMAS [**]
[Priority: 1]
```

The [**] is not relevant to the alert, but makes it easy to see where the main information about the attack is in the alert file. Now let's see what the different numbers stand for:

The first number is the Generator ID; this tells the user what component of SNORT generated this alert. For a list of GIDs, we can refer to Table 1. In this case, the alert was made by the `rules_subsystem` (1) component of SNORT.

The second number is the SNORT ID (also referred to as Signature ID). For a list of preprocessor SIDs, `gen-msg.map` file should be seen. The SID number is written directly into the rule by using the `sid` option. In this case, 1228 represents a nmap XMAS scan.

The third number is the revision ID. This number is primarily used when writing rules, as each revision of the rule should increment this number with the "rev" option. In this example we see that this is the 7th revision of this rule.

At the beginning of the text string, there is a word with only uppercase letters. This word tells the user where to find the rule. In this case the rule is from the `scan.rules` file in `/etc/snort/rules/` (on a Unix system). The rest of the fist line is the message telling what the alert is alerting on.

The second line is telling the administrator how serious the alert is. SNORT uses either the `classtype` option or the `priority` option to give this information. Rules that have a classification will have a default priority set (from 1-4, where 1 is the most severe). The classifications used by the rules provided with SNORT are defined in etc/classification.config on a Unix system. When using the `priority` option, the rule maker can decide on what number to use. We will use 0 as the number for our rules, to make them stand out from the rest.

Table 1: A selection of Generator ID's

Generator name	GID	Comment
rules_subsystem	1	SNORT Rules Engine
tag_subsystem	2	Tagging Subsystem
portscan	100	Portscan1
http_decode	102	HTTP decode 1/2
bo	105	Back Orifice
unidecode	110	unicode decoder
stream4	111	Stream4 preprocessor
decode	116	SNORT Internal Decoder
scan2	117	portscan2
sfportscan	122	Dan Roelkers portscan
frag3	123	Marty Roesch's ip frag reassembler

5.2 Honeypots

5.2.1 Definition of a honeypot

There are many definitions of a honeypot. They depend on the people using them, and what they want to accomplish. Here are three possible ways of defining a honeypot, one just as correct as another:

- a solution to lure or deceive attackers
- a technology used to detect attacks
- real computers designed to be hacked into and learned from

Spitzner [41, 42] gives the following two definitions: "A honeypot is a resource whose value is in being attacked or compromised. This means, that a honeypot is expected to get probed, attacked and potentially exploited. Honeypots do not fix anything. They provide us with additional, valuable information" and "A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource".

What these definitions tell us is that honeypots are not limited to solving only one problem. They have a wide range of application. Basically, there are two different categories of honeypots: production and research. Research honepots are used to collect information from the network, while production honeypots are used to protect the network and secure the organization. We will use a honeypot called `honeyd` as a research honeypot in this thesis for the purpose of collecting information about attacks.

Honeypots can generally be divided into three different categories, low-interaction, medium-interaction and high-interaction honeypots [41]. A high-interaction honeypot simulates all aspects of an operating system. A low-interaction honeypot simulates only some parts, for example the network stack. A high-interaction honeypot can be compromised completely, allowing an attacker to gain full access of the system and use it to launch further attacks. In contrast, low-interaction honeypots only simulate services that cannot be exploited to get complete access of the honeypot. Low-interaction honeypots are more limited, but they are useful to gather information at a higher level, e.g., learn about network probes or worm activity. Medium-interaction honeypots fall somewhere in between its low and high interaction counterparts in that they are home-made and not some out-of-the-box pre-made solution. Medium-interaction honeypots can range from a simple port listener to a complete host just sitting on a network waiting to be attacked. Basically, these honeypots are built and completely customized by those who will be administering them.

Honeypots may also be divided into two broad categories; production and research. The purpose of a production honeypot is to help mitigate risk in an organization. The honeypot adds value to the security measures of an organization. A production honeypot is usually synonymous with a low-interaction honeypot. This is because it is the safest interaction type to deploy in production networks. The second category, research, includes honeypots, which are designed to gather information on the community of attackers. These honeypots are used to gather intelligence on the general threats organizations may face, allowing the organization to better protect against those threats. Usually a medium- or high-interaction honeypot is used for this purpose, but also low-interaction honeypots such as `honeyd` could be used. The reason for `honeyd` to fall into this category is because of its sophisticated way of dealing with requests, as discussed below.

5.2.2 `Honeyd`

`Honeyd` [34, 35] is a low-interaction honeypot used to simulate virtual hosts on a network. The main use of `honeyd` is in honeynet research, typically for setting up virtual honeypots to engage attackers. It can fake the personality of "any" operating system, and can be configured to offer different TCP/IP "services" like HTTP, SMTP, SSH, FTP etc. These emulated services make it possible to determine what the attackers are attempting to do and what they are looking for. This is done by creating scripts that listen on specific ports and then interact with attackers in a predetermined manner. We use the wuftpd.sh and apache.sh scripts in the experiments to gather information about web and FTP attacks.

A very useful feature of `honeyd` is its ability to simulate an entire network within one machine. It is even possible to define factors as hops, packet loss and latency. This lets us simulate networks in the test lab and present a virtual network to an attacker.

5.2.3 Configuring `honeyd`

The following section is written based on the article [37]. A virtual honeypot is in `honeyd` configured with a template created in a configuration file (`honeyd.conf`) that defines the characteristics of a honeypot (OS type, ports it listens on and behaviour of emulated services). Each template is given a name (i.e. windows, linux, default). A new template is created by using the `create` command. The `set` command assigns a personality from a NMAP fingerprinting file to the template. This personality determines the network behaviour of the given operating system that is simulated by `honeyd`. The `set` command also defines the default behaviour for network protocols: reset, open or block. When using block, all packets for the specified protocol are dropped by default. If using reset, the ports are closed by default. Open means that all ports are open by default. Another important command is `add`. This command is used to specify the services that are remotely accessible. `Bind` is used to assign a template to an IP address. Sample scripts using these commands are available in various technical reports (e.g., [16, 20], and in Appendix A of this thesis.

5.2.4 Honeyd log files

`Honeyd` logs information about attacks in different log files depending on which service the attack was pointed at. In the experiment chapter (chapter 6), we make use of two of the honeypot's log files. These files are `web.log` and `honeyd.txt`. In conjunction with these two files we also include the "main" log file `honeyd.log` in the explanation below.

The structure of honeyd.log

All connections to and from the honeypot are logged in a file named `honeyd.log`. The `honeyd.log` file consists of the following fields:

<date & time><protocol><seebelow><src_IP><src_port><dst_IP><dst_port:><packet_size>

<flags><OS fingerprint>

The third field may either be S, E or -. S means the start of a new connection, E the end of a connection and - if a packet is not belonging to any connection. On lines with E, `honeyd` logs the amount of data received and sent at the end of the line. An example of three `honeyd.log` entries are shown in Figure 2.

```
2006-03-22-13:13:27.4148 tcp(6) - 128.39.44.11 54265 128.39.44.40 80: 40 R [Linux 2.6 ]
2006-03-22-13:13:27.4959 tcp(6) S 128.39.44.11 55286 128.39.44.40 80 [Linux 2.6 ]
2006-03-22-13:13:32.5889 tcp(6) E 128.39.44.11 55286 128.39.44.40 80: 48 1058
```

Figure 2: Example of three `honeyd.log` entries

The structure of honeyd.txt

This log file is used by a number of service scripts like:

- msftp.sh
- wuftpd.sh
- ssh.sh
- telnetd.sh

In our approach to automate rule generation, we suggest that each service logs to its own file because this makes it easier to determine what type of service was attacked. If all services were to log to the same file, it would have been more difficult to sort the entries in the file based on services. Another solution would be to make a script that sorts entries based on the –MARK– line as shown below. This line gives information about destination and source port number. As the example (Fig. 3) shows, each element of an attack is logged between –MARK– and –ENDMARK–. In this particular attack we only need the line with stat ../* to make the rule out of. As discussed earlier in this section we also might need the first line to be able to determine what type of service was attacked.

```
--MARK--,"Fri Apr 14 14:01:07 CEST 2006","wu-ftpd/FTP","128.39.44.11",
"128.39.44.55",42512,21,
"USER anonymous
PASS nessus@nessus.org
stat ../*
PASV
",
--ENDMARK--
```

Figure 3: Example of a `stat ../*` attack

The structure of web.log

`Web.log` is used by the `apache.sh` script which is run on some of the virtual hosts on the honeypot. The following (Fig. 4) is an example from the web.log after an attack has occurred on a virtual SuSE host. The only information we want is to know what the attack consisted of and what it was attacking. The first line gives us the information that an attack against our apache script on port 80 has occurred. Also the source IP and source port are logged on this line. On the following line the actual attack is logged. All lines below this line are not important, and are removed before the rule generating is started.

```
--MARK--,"Tue Apr 11 13:47:50 CEST 2006","apache/HTTP","128.39.44.11","128.39.44.50",33537,80,
"GET /login.html HTTP/1.1
Connection: Close
Host: 128.39.44.50
Pragma: no-cache
User-Agent: Mozilla/4.75 [en] (X11, U; Nessus)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
",
--ENDMARK--
```

Figure 4: Example of an attempt to login on the web page

5.3 Nessus

In this section we explain how the vulnerability scanner or a so called penetration testing utility called Nessus [4] works. We need a vulnerability scanner in order to control the attacks toward our honeypot, make it possible to reproduce the experiment results and to get a dataset containing several different attacks. Nessus' value is to scan networks for vulnerabilities and report if any are found. We use Nessus to scan the honeypot in order to get a dataset containing malicious traffic. There are a number of free security scanners available: SAINT [6], VLAD [10] and Nessus are only a few of them. In [15], Bauer states that *Nessus stands out as a viable alternative to powerful commercial products, such as ISS' Internet Scanner and NAI's CyberCop Scanner.* He also states that Nessus in many ways exceeds the usability and flexibility of its proprietary counterparts. Also an open source program named Metasploit [3] has been developed to test IDS'. This program was too difficult to use, and lacked the possibility to automate the scan in a simple manner. Because of the before mentioned flaws in other scanners, Nessus is our choice.

Nessus is a vulnerability scanner that tells the user what security flaws are present in the system. Nessus depends on nmap to be able to do its tasks. We use Nessus because it is a widely deployed vulnerability scanner both amongst administrators and crackers.

Once Nessus has determined which services are present (by using nmap), it performs various checks to determine which software packages are running, which version they are and whether they're subject to any known vulnerabilities. Predictably, this level of intelligence requires a good vulnerability database that must be updated periodically as new vulnerabilities come to light. We only updated the database at installation to avoid the probability of getting different results during the experiments. If Nessus does not include a vulnerability check for something we would like to have, it is possible to write our own vulnerability checks. This is not something all security scanners have, but Nessus does and hence gives us the possibility to customize it after our needs. When Nessus has finished a scan, it reports all vulnerabilities found and explains them in detail and how to fix them.

Nessus' Architecture

Nessus consists of two major parts: a server (nessusd) which runs all the scans, and a client, with which one can control scans and view reports. This distributed architecture makes Nessus flexible and also allows us to mix platforms if necessary. This is not something we probably need, since we can run without problems both the server and client on the same machine. When the client is connected to the server, a list of plug-ins (vulnerability tests) supported by the server and a number of other options appear. (How we configure Nessus is explained in more detail in the experiment chapter). Once a scan

is initiated, Nessus invokes each appropriate module and plug-in as specified, beginning with the nmap scan (if required). The results of one plug-in test determine if a subsequent test should be run or how. When the scan is finished, the results are sent back to the client. We only utilize this report to ensure that the scan was successful in scanning the service we specified.

5.4 How to generate SNORT rules

In this section we explain how we generate the rules used in the experiments. Before the rule generating can begin, we need a dataset with malicious traffic. This is taken care of by the honeypot and Nessus. We use Nessus to scan a specific service on a specific virtual host on the honeypot. The traffic is also run through the `SNORT` IDS to see what traffic raises alerts. Then the dataset is edited to only include data `SNORT` alerted on. This is because we only want to create rules for traffic we know `SNORT` has a rule for. The reason for this is that we need to have a counterpart in order to measure the differences between the original rules and the new rules. The goal is to se if we are able to create working rules based on information logged by a low-interaction honeypot as `honeyd`. We will compare the new rules to the originals by measuring the differences based on performance (False positives/False negatives), in addition to ranking each missing field by their importance as we see it. A procedure for extracting rules from honeypot log files is presented in the experiment chapter for both web attacks and FTP attacks. An important part of the rule generating was to make it as automatic as possible, using only information given by the honeypot and the standard way of writing `SNORT` rules. We had to use some assumptions in the procedure regarding what fields `SNORT` most likely would use for the attacks we deploy. This does not influence the results because the knowledge we have of `SNORT` rules are freely available to anyone wanting to implement our procedure into a working program. How our procedure would perform on traffic not made by Nessus is a different question, but in this thesis we will only use this traffic.

All rules are generated by hand following the procedures presented in each of the two experiments, bearing in mind that only information from the honeypot may be used. By definition, all traffic towards a honeypot is malicious, hence all the possibilities of creating rules, which result in false positives are greatly reduced.

5.5 Longest Common Substring algorithm

We use the LCS algorithm [22] to reduce the number of rules created. By using this algorithm it is possible to create one rule for several similar attacks. This is important because it is a relation between `SNORT`'s processing speed and the number of rules it loads.

The LCS algorithm [11, 22] is used to find the longest string(s) that is a substring or are substrings of two or more strings. There are several ways of implementing this algorithm, such as using suffix trees or dynamic programming (matrix). We chose to use the latter because our strings are short, hence the computational overhead is not so important. It is also the easiest to understand. The problem though is that the LCS is not suited for polymorphic worms [31]. This is because polymorphic worms change too much of its payload for LCS to get a good result out of. To cope with this, another similar algorithm may be used, namely the Longest Common Subsequence [11, 38, 22]. We chose not to use this algorithm because our traffic does not include any polymorphic attacks.

5.6 True/False Positive Ratio

True Positive Ratio (TPR) is a way of showing how good the IDS is at alerting on real attacks. In our setting we use this to show how good our rules are compared to the

originals. TPR is obtained by the following formula:

$$TPR = \frac{TP}{TP + FN}$$

Where: TP = The number of alerts on malicious traffic, FN = The number of missing alerts on malicious traffic. The total number of intrusions is given by $TP + FN$.

False Positive Ratio (FPR) shows the proportion of instances, which were not an attack but still were alerted on. FPR is a result of the following formula:

$$FPR = \frac{FP}{FP + TN}$$

Where: FP = The number of alerts on benign traffic, TN = The number of correct decisions on benign traffic. The total number of no-intrusions is given by $FP + TN$.

A perfect IDS would have TPR = 1 and FPR = 0. This would result in alerts only on malicious traffic, and no alerts on benign traffic.

The confusion matrix in Fig 5 illustrates what FP, FN, TP and TN mean.

Attack?	Alert? YES	Alert? NO
YES	TP	FN
NO	FP	TN

Figure 5: Confusion matrix

5.7 Improved system for datacollection

The following section is about a data collection system which involves the use of anomaly detection, IPS and honeypots. This system has not been tested or given a thorough study. It is only meant as a suggestion of how to improve the amount of data collected by the honeypot. Because of the fact that a honeypot only sees traffic directed towards it makes our system vulnerable to attacks directed directly to the production network. We have tried to solve this problem by designing a system based on previous work related to honeypots and IDS'. In this section we present a rule generating system based on work done by other authors. The general idea for data collection is based on an article written by Anagnostakis et al. [13]. We use the shadow honeypot system in conjunction with an anomaly detector to direct malicious traffic towards the honeypot. This is useful since a normal honeypot only interacts on traffic destined towards it. The major problem with this is that the honeypot must be detected by the attacker, and then the attacker must direct the attack towards it. If the attacker does not find the honeypot interesting he/she might attack other computers on the network instead, making us unable to log the attack in the way a honeypot can. But with the use of an anomaly based IDS, traffic destined to all hosts on the network can be re-directed towards the honeypot if it is beleived to be malicious. It is then important that the honeypot is able to mislead the service requested in such a manner that the attacker does not see the difference. Anagnostakis et al. [13] solve this by having a mirrored version of the real service on the honeypot. The honeypot then determines if the request is benign or not, if it is, the traffic is redirected

to the normal service and dealt with as normal. In our approach we will use a normal honeypot (preferably a low-interaction one if it is feasible to create scripts good enough in order not to be revealed by the attacker) instead of the shadow honeypot. The reason for us to prefer the low-interaction honeypot is because this type of honeypot does not need as much maintenance and monitoring as high-interaction needs. It is also a goal to keep it as simple as possible to make it usable to people with limited knowledge of such systems. Sadly we did not have the time to implement this system, and it is therefore presented as a possible future work in section 7.1. A flowchart of our presented system is shown in figure 6.

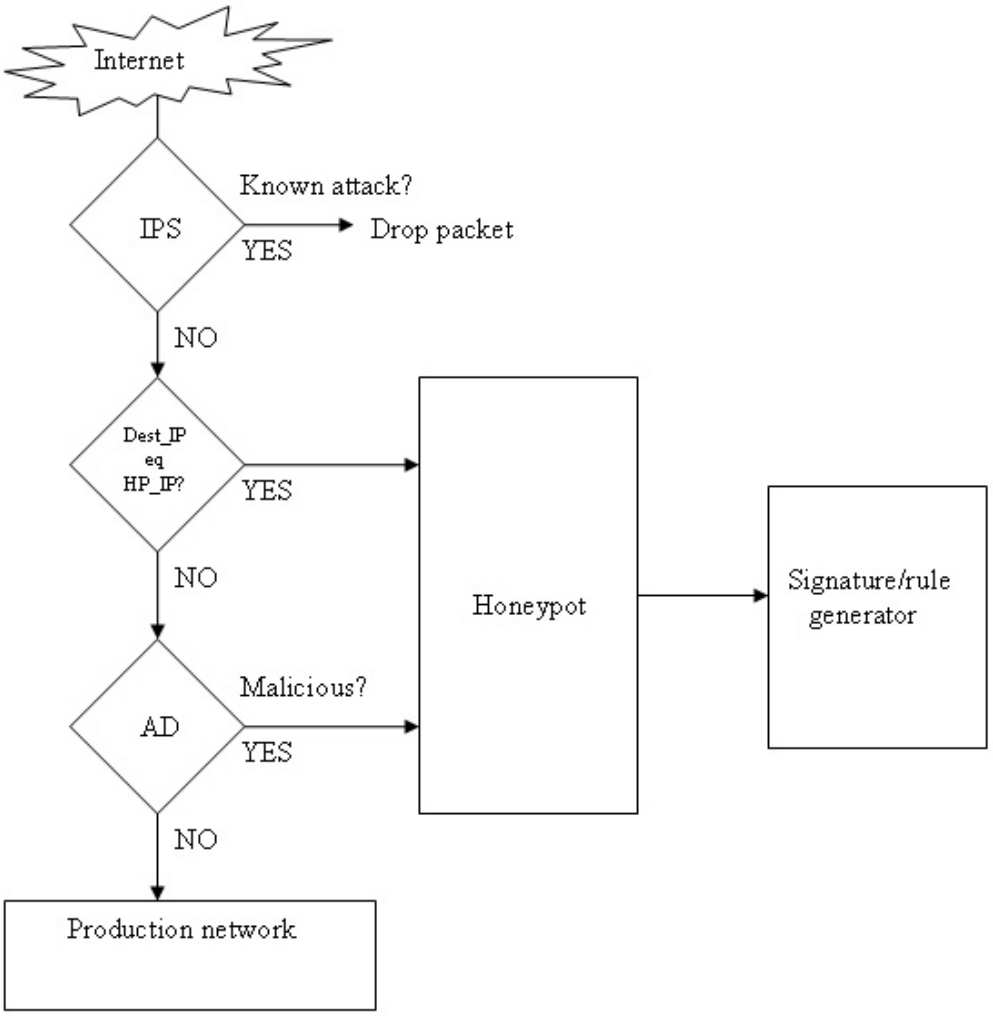

Figure 6: Flowchart of rule generating system

Explanation to figure 6:

- The IPS is an Intrusion Prevention System (Snort-inline)
- Dest_IP eq HP_IP is only included to visualise that traffic destined to the honeypot is of course going to the honeypot. This is not something necessary to implement, since this is done automatically
- AD is an anomaly detector used to re-direct possible malicious traffic to the honeypot
- The honeypot interacts with the possible malicious traffic and logs as much about the

request/attack as possible

- The signature/rule generator generates rules and signatures based on the logging information from the honeypot

The data collected by the honeypot will be analyzed by the signature generating system in order to create IDS rules.

6 Experimental work

All experiments in this chapter use traffic produced by Nessus with an updated plug-in database (as of 25th of March 2006). The reason for this is because our preliminary experiment showed that the honeypot did not get enough different attacks to create a dataset from. For the FTP experiment, the Nessus FTP plug-in was used for scanning. For the web server scan, the Nessus web server plug-in was used. Nessus was configured to only do one scan at a time because this would produce traffic in the same order each time. It was also configured to only scan one port number (for FTP -> port 21, for web -> port 80). This reduces the amount of unwanted traffic in the log files. Each experiment was run on all the different OS types, which included the specific service script. For FTP this means that host 128.39.44.33, 128.39.44.50 and 128.39.44.55 were scanned. The same was done for the web server scan. But only the host, which triggers the highest amount of different alerts, is used in the control experiment. During all the scanning attempts, `SNORT` is used to collect alerts. All occurrences of an alert are counted, before the alert file is edited to only include one alert from each of the triggered rules (often one rule is triggered several times). Then the corresponding rule to each alert was included into the file to make it easier to see what the alert alerted on. When this is done, the alert/rule file is compared to the log file from the honeypot (For FTP, `honeyd.txt`. For WEB, `web.log`). All the rules are then reproduced as well as they can be, based on the honeypot log. The goal then is to measure the differences between the original and the new rule. When the new rules have been created, the experiment is repeated with the new rules implemented in `SNORT`. Then the alert file from the control experiment is compared to the new experiment. Hopefully the new rules will trigger on the same traffic as the originals do.

6.1 Strategy

Before the experiments can be run, it is important to be aware of all parameters able to affect the results of the experiment.

Reliability: To ensure reliability and reproducibility in the experiments, we have done the following:

1. Always use Nessus as a traffic generator, with the same options set each time
2. Providing a full description of how the software has been installed (Appendix A, version numbers and hardware setup (6.2)
3. Document how the scan is executed

Type of scan: We have used Nessus as a traffic generator. The reason for this is that Nessus has a wide range of scans available, it is free of charge, widely used for testing IDS' and also used by hackers to scan networks. We have used Nessus to scan our virtual hosts with only one port at a time (horizontal scan), to be able to see which hosts give the highest amount of different `SNORT` alerts. This is because we want to have as large dataset as possible to give the experiments more validity. We ended up using the `web servers scan` and the `FTP scan`. Another reason for us to use a security scanner instead of leaving the honeypot on a wild network is because of the risks involved with this. A skilled cracker might be able to "steal" our honeypot, and use it to launch attacks against other hosts on our network or other networks. By using a traffic generator like Nessus it is also easier to reproduce the experiment scenarios later.

Speed of the scan: To make sure the honeypot catches all the attacks sent by Nessus we have configured Nessus to only do one scan at a time. This also serves another purpose, giving us the same order of attacks in the log files, making it much easier to compare the results.

Honeypot type: We chose to use a low-interaction honeypot because of the security aspects regarding high-interaction honeypots [41]. In our experiments this would not have been a problem since we don't use any live traffic, but if our procedure is to be used for discovering new attacks the honeypot must be deployed in the wild.

Presentation: How to present the results is a much debated topic. The most frequently used method to graphically represent FPR and TPR for IDS' are ROC curves, even though these curves do have some weaknesses [28]. There exist a few examples on how to modify these curves to fit the IDS evaluation [29]. We will present the results using tables and ratios.

Changing the configuration of the honeypot or IDS: Altering some of the configuration options in `SNORT` may make it alert more/less often. We have agreed upon keeping the `SNORT` configuration file as it is by default. The only alterations we make is what rule files to load. This depends on what experiment is run. The configuration file used by the honeypot is kept static during the experiments. A test experiment is run before the real ones to make sure everything works. We also made a perl script to erase all log files before `honeyd` starts to make sure everything is "clean".

6.2 Overview and technical information

In this section we show how the hosts we use are connected, and what their specifications are.

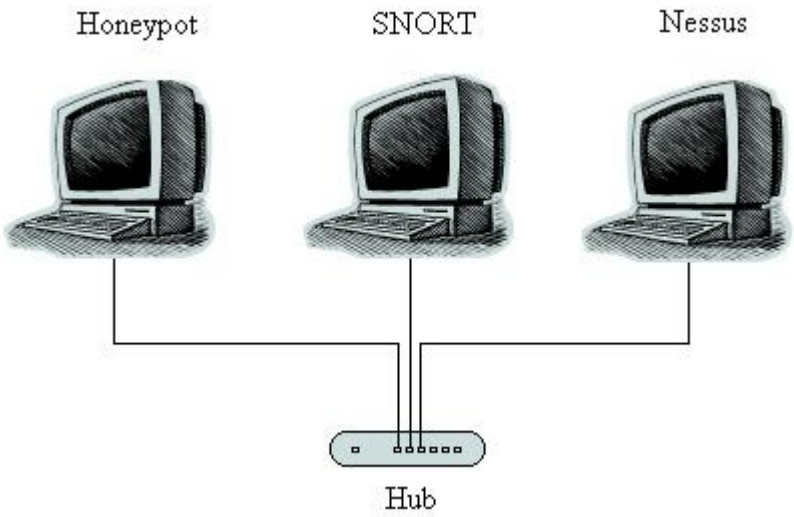

Figure 7: Network topology

The Nessus host

```
Dell Optiplex GX150 (2001)
* 1 GHz Pentium III
```

```
* 256 MB RAM
* 40 GB HD

OS = Linux ubuntu 2.6.10-10-386, fully updated
Nessus v2.2.4 for Linux, with updated plugin database as of 03.28.06
```

The honeypot

```
Dell Optiplex GX150 (2001)
* 1 GHz Pentium III
* 256 MB RAM
* 40 GB HD

OS = Linux Fedora Core 3, 2.6.12-1.1381.FC3
Honeyd v1.0a
```

The SNORT host

```
Dell Optiplex GX150 (2001)
* 1 GHz Pentium III
* 256 MB RAM
* 40 GB HD

OS = Linux ubuntu 2.6.12-9-386, NO updates
SNORT v2.3.2 (build 12) with rules updated 03.30.06
```

The hub

```
HP advanceStack Hub-8U, 10Base-T, HPJ2610B
```

6.3 Experiment result expectations

We do these experiments to see if the following hypothesis stands:

Hypothesis: honeyd logs sufficient data about attacks to re-create SNORT rules

This hypothesis has two sides. First, we want to see if honeyd logs enough data to re-create a known SNORT rule and that triggers properly on the same traffic pattern as the original. Second, we want to see if it is possible to extract important rule header and rule option fields for SNORT rules from the honeyd logs. We expect that this hypothesis will stand for the majority of rules we re-create because of the amount of data honeyd logs. A quick study of the log files shows that a great deal of information is logged for each attempted attack. So the question is if it logs enough data to re-create known SNORT rules in such a manner that they work satisfactorily.

6.4 Measurement method

Each field from the SNORT rule is ranked by a number from 0-6, where 6 is the most important and 0 is the least important (Table 2). We will only include fields used in the original rules, which were triggered in the control experiment (Table 3). It is important to note that even though a field gets the rank of one, this does not mean that the field is needless to include. This ranking is based on what fields are important to make the rule work with as few false positives and false negatives as possible.

Table 2: Importance of ranking numbers

Ranking	**Comment**
0	Systematic error. Meaning that this is excluded from the experient because it is impossible to automate the making of this field based on the honeypot logs
1	Not important
2	Important for the rule to make sense to the administrator
3	Important for performance
4	Important for performance and correct triggering
5	Important for performance, triggering and reducing FNR & FPR
6	Vital for the rule to work

Table 3: Ranking of rule fields

Field name	Ranking	Comment
protocol	6	Must be specified
src_adr	6	Use any or $EXTERNAL_NET
dst_adr	6	Name the host in the rules and in the SNORT config file, i.e. $HTTP_SERV. This makes SNORT work faster because it now can skip checking the traffic against rules which don't match the destination address
src_port	6	Use any
dst_port	6	Same as for dst_adr
msg	2	Needed for the administrator to understand what triggered the rule
flow	4	Important for performance because of the direction of the attack
content	6	Vital to make the rule trigger
uricontent	6 (5)	A 5 when used togheter with content
distance	5	Specifies how far into a packet SNORT should ignore before searching for a pattern. Similar to depth, but distance is relative to the last pattern match instead of the beginning of the packet
pcre	5	Perl regular expression, makes it possible to trigger on slightly different versions of the same attack
within	4	maximum of N bytes between two Content keywords
depth	4	How deep into the packet to search
nocase	4	To be able to trigger on all case mixes of the same attack
reference	0	Only value is for admin to be able to find places to read about the attack. Not considered to be important because this information can easily be found on google. Zero-day attacks does not have any available references
classtype	2	Without this, priority always equals zero in the alert
Priority	2	Used instead of classtype to set the priority directly
sid	1	Not important
rev	1	Not important

6.4.1 Measuring known signature generating systems

Earlybird

Earlybird [39] is a signature genertaing system for worms. The only example found of a rule created by Earlybird is as follows:

```
drop tcp $HOME_NET any -> $EXTERNAL_NET 5000 (msg:"2712067784 Fri May 14 03:51:00 2004";
rev:1; content:"|90 90 90 90 4d 3f e3 77 90 90 90 90 ff 63 64 90 90 90 90 90|";)
```

We were not able to find a corresponding original SNORT rule to compare with, but a qualified guess would be that sid, classtype and flow would have been included. Also the

msg field is not telling the administrator anything about the attack, which we consider as important.

Honeycomb
Honeycomb [26] is another attempt on making IDS rules automatically. We have found two outputs from this generator, and will assess these two now. First off is a rule made to alert on the Slammer worm:

```
alert udp any any -> 192.168.169.2/32 1434 (msg: "Honeycomb Fri Jul 18 11h46m33 2003 ";
content: "|04 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 DC C9 B0|B|EB 0E 01 01 01 01 01 01 01|p|AE|B
|01|p|AE|B|90 90 90 90 90 90 90 90|h |DC C9 B0 |B|B8 01 01 01 01|1|C9 B1 18|P|E2 FD|5
|01 01 01 05|P|89 E5|Qh.dllhel32hkernQhounthickChGetTf|B9|llQh32.dhws2_f|B9|etQhsockf
|B9|toQhsend|BE 18 10 AE|B|8D|E|D4|P|FF 16|P|8D|E|E0|P|8D|E|F0|P|FF16|P|BE 10 10 AE|B|
8B 1E 8B 03|=U|8B EC|Qt|05 BE 1C 10 AE|B|FF 16FF D0|1|C9|QQP|81 F1 03 01 04 9B 81 F1 01
01 01 01|Q|8D|E|CC|P|8B|E|C0|P|FF 16|j|11|j|02|j|02 FF D0|P|8D|E|C4|P|8B|E|C0|P|FF 16
89 C6 09 DB 81 F3|<a|D9 FF 8B|E|B4 8D 0C|@|8D 14 88 C1 E2 04 01 C2 C1 E2 08|)|C2 8D 04
90 01 D8 89|E|B4|j|10 8D|E|B0|P1|C9|Qf|81 F1|x|01|Q|8D|E|03|P|8B|E|AC|P|FF D6 EB|"; )
```

Next is the rule created for the Code Red II worm:

```
alert tcp 80.0.0.0/8 any -> 192.168.169.2/32 80 (msg: "Honeycomb Mon May 5 16h59m09 2003
"; flags: A; flow: established; content: "u|08 81|.0|9A 02 00 00 0F 84 C4 00 00 00
C7|F0|9A 02 00 00 E8 0A 00 00 00|CodeRedII|00 8B 1C|$|FF|U|D8|f|0B C0 0F 95 85|8|FE FF
FF C7 85|P|FE FF FF 01 00 00 00|j|00 8D 85|P|FE FF FF|P|8D 85|8|FE FF FF|P|8B|E|08 FF|
p|08 FF 90 84 00 00 00 80 BD|8|FE FF FF 01|thS|FF|U|D4 FF|U|EC 01|E|84|i|BD|T|FE FF
FF|,|01 00 00 81 C7|,|01 00 00 E8 D2 04 00 00 F7 D0 0F AF C7 89|F4|8D|E|88|Pj|00 FF|u|08
E8 05 00 00 00 E9 01 FF FF FF|j|00|j|00 FF|U|F0|P|FF|U|D0|Ou|D2 E8|;|05 00 00|i|BD|T|FE
FF FF 00|\&|05 81 C7 00|\&|05|W|FF|U|E8|j|00|j|16 FF|U|8C|j|FF FF|U|E8 EB F9
8B|F4)E|84|jd|FF|U|E8 8D 85|<|FE FF FF| P|FF|U|C0 0F B7 85|<|FE FF FF|=|88 88 00 00|s|
CF 0F B7 85|>|FE FF FF 83 F8 0A|s|C3|f|C7 85|p|FF FF FF 02 00|f|C7 85|r |FF FF FF
00|P|E8|d|04 00 00 89 9D|t|FF FF FF|j|00|j|01|j|02 FF|U|B8 83 F8 FF|t|F2
89|E|80|j|01|Th.f|04 80 FF|u|80 FF|U|A4|Yj|10 8D 85|p|FF FF FF|P|FF|u|80 FF|U|B0 BB 01
00 00 00 0B C0|tK3|DB FF|U|94|=3'|00 00|u?|C7 85|h|FF FF FF 0A 00 00 00 C7 85|l|FF FF
FF 00 00 00 00 C7 85|'|FF FF FF 01 00 00 00 8B|E|80 89 85|d|FF FF FF 8D 85|h|FF FF
FF|Pj|00 8D 85|'|FF FF FF|Pj|00|j|01 FF|U|A0 93|j|00|Th.f|04 80 FF|u|80 FF|U|A4|Y|83
FB 01|u1|E8 00 00 00 00|X-|D3 03 00 00|j|00|h|EA 0E 00 00|P|FF|u|80 FF|U|AC|=|EA 0E 00
00|u|11|j|00|j|01 8D 85|\|FE FF FF|P|FF|u|80 FF|U|A8 FF|u|80 FF|U|B4 E9 E7 FE FF FF BB
00 00 DF|w|81 C3 00 00 01 00 81 FB 00 00 00|xu|05 BB 00 00 F0 BF|'|E8 0E 00 00 00
8B|d$|08|dg|8F|"; )
```

As both these examples show, Honeycomb only includes rule options that are necessary to make the rule trigger. Both examples are without `priority, sid, reference` and `rev` (these options are easy to automate, but still they have chosen to leave them out). As it seems, this rule generator is good at extracting signatures from the honeypot, but not very good at creating "complete" SNORT rules.

6.5 FTP scan experiment

6.5.1 How the experiment is conducted

The virtual host on the honeypot which gives the highest amount of different alerts is used in this experiment (a Linux host running the wuftpd.sh script). This virtual host is then attacked by Nessus' FTP scan. Another machine running SNORT is checking the traffic produced by Nessus and the virtual host, and giving alerts on traffic it finds malicious.

6.5.2 Experiment 1 - Control

The following information is gathered from the log file where `honeyd` logs ftp attacks (`honeyd.txt`). Honeyd logged this information after a Nessus FTP scan against one of the honeypot's virtual hosts. In this particular case it was a Linux host with the `wuftpd.sh` script, which was attacked. Only the strings with a corresponding rule are listed here. The actual log file consists of 353 strings.

- RETR .rhosts
- stat ../*
- SITE exec /bin/sh -c bin/id
- SITE exec /bin/sh -c /usr/bin/id
- CWD ..%20
- CWD ..%20
- CWD ~nonexistinguser
- RETR .forward
- CWD ~root
- SITE EXEC %p
- CWD ~root

6.5.3 Experiment 2 - New FTP rules

Based on the information from the section above, we want to know how good rules we can make. We will use the measurement method from subsection 6.4 to make some statistics showing the quality of the new rule. The following experiment is conducted in the same manner as the control experiment above. The only difference is that we now switch the original rules with the new rules we have created.

FTP scan against a virtual Linux machine with a wuftpd.sh script

In this section we will try to create SNORT rules based on the strings from Section 6.5.2. The goal is to see if we are able to make useful rules based on information from the honeypot. We will compare the new rules to the originals by measuring the difference based on the ranking system found in Section 6.4.

Before the process of creating rules started, the honeypot logfile was edited to remove strings which were irrelevant. These string were:

- –MARK–,"Mon Apr 3 19:09:20 CEST 2006","wu-ftpd/FTP","128.39.44.11","128.39.44.55",60699,21,
- –ENDMARK–
- USER
- PASS
- PASV
- QUIT
- ",
- SYST
- CWD /

Procedure for creating FTP rules

We now present a new algorithm to automate FTP rule generation:

```
int $count = 1000000;  \\ used for sid numbering
int $X = 5;            \\ used for LCS minimum output
string $cont;          \\ storing content (signature)

Use "grep" to remove irrelevant data in the log file.
Sort the file alphabetically.
If many lines starting with equal content (3 or more), use LCS to
find what is equal. (A LCS result should never be shorter than X characters,
this is because a to short string might trigger on numerous strings it has
```

```
nothing to do with).

set X;  \\ Promt user for minimum LCS output

while (honeyd.txt not empty){
$cont="string from file"

print to rulefile {
alert tcp $EXTERNAL_NET any -> $HOME_NET $FTP_PORTS (msg:"NEW FTP $cont";
flow:to_server,established; content:"$cont"; nocase; priority:0;
sid:$count; rev:1;)

$count ++;
else
ERROR
}
```

This procedure is an attempt to automate the creating of the rules. All fields in the alerts created are static because they were in use in all the original files we tried to recreate. The information for `flow` and `protocol` is available in the honeypot file `honeyd.log`, and how to extract this information and interpret it is presented in section 6.7. We also use `nocase` in all of our rules to make sure `SNORT` ignore the case of the letters. This may lead to false positives, but this is highly unlikely. Since the `honeyd.txt` file only includes FTP attacks in our experiment, we do not have to do any checks to see if it really is a FTP attack. If deployed in the wild, each service would log to its own file. The information needed to distinguish what service the attack was meant for is also available in the `honeyd.txt` file, so it is possible to log everything in one file if desired. Using a database to store the attacks in is also a possibility.

The rules we made by hand from the honeypot log, are shown below. These rules have been made following the procedure above (section 6.5.3. The number inside the [] is the SID number of the original rule.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP RETR .rhost";
flow:to_server,established; content:"RETR .rhost"; nocase; priority:0;
sid:1000000; rev:1;)
```

Different from the original: [335]

- content:".rhost";
- nocase not supposed to be included
- no reference
- no classtype:suspicious-filename-detect

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP stat ../*";
flow:to_server,established; content:"stat ../*"; nocase;  priority:0;
sid:1000001; rev:1;)
```

Different from the original: [1777]

- content:"STAT";
- no pcre:"/^STAT\s+[^\n]*\x2a/smi";

- no reference
- no classtype:attempted-dos

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP SITE exec
/bin/sh -c /usr/bin/id"; flow:to_server,established;
content:"SITE exec /bin/sh -c /usr/bin/id"; nocase;  priority:0;
sid:1000002; rev:1;)
```

Different from the original: [361]

- content:"SITE";
- content:"EXEC";
- no pcre:"/^SITE\s+EXEC/smi";
- no distance:0
- no reference
- no classtype:bad-unknown

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP SITE exec
/bin/sh -c /bin/id"; flow:to_server,established;
content:"SITE exec /bin/sh -c /bin/id"; nocase;  priority:0;
sid:1000003; rev:1;)
```

Different from the original: [361]

- content:"SITE";
- content:"EXEC";
- no pcre:"/^SITE\s+EXEC/smi";
- no distance:0
- no reference
- no classtype:bad-unknown

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP CWD ..%20.";
flow:to_server,established; content:"CWD ..%20."; nocase;  priority:0;
sid:1000004; rev:1;)
```

Different from the original: [360]

- content:".%20."
- no reference
- no classtype:bad-unknown

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP
CWD ~nonexistinguser"; flow:to_server,established;
content:"CWD ~nonexistinguser"; nocase;  priority:0; sid:1000005; rev:1;)
```

Different from the original: [1672]

- content:"CWD";

- no pcre:"/^CWD\s+~/smi";
- no reference
- no classtype:denial-of-service

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP RETR .forward";
flow:to_server,established; content:"RETR .forward"; nocase;  priority:0;
sid:1000006; rev:1;)
```

Different from the original: [334]

- content:".forward";
- nocase not supposed to be included
- no reference
- no classtype:suspicious-filname-detect

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP CWD ~root";
flow:to_server,established; content:"CWD ~root"; nocase;  priority:0;
sid:1000007; rev:1;)
```

Different from the original: [336, 1672]

- content:"CWD"; nocase; content:"~root"; distance:1; nocase; [336]
- no pcre:"/^CWD\s+~root/smi"; [336]
- content:"CWD"; [1672]
- no pcre:"/^CWD\s+~/smi"; [1672]
- no reference
- no classtype:bad-unknown

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"NEW FTP SITE EXEC %p";
flow:to_server,established; content:"SITE EXEC %p"; nocase;  priority:0;
sid:1000008; rev:1;)
```

Different from the original: [361]

- content:"SITE";
- content:"EXEC";
- no pcre:"/^SITE\s+EXEC/smi";
- no distance:0
- no reference
- no classtype:bad-unknown

Results from testing the new rules compared to the originals

In this experiment we have added the new rules into SNORT. Then we have done exactly the same as we did in the control experiment.

Alert statistics overall:

- False negatives: 0

Table 4: Number of alerts from each rule when a Nessus FTP scan is run

Nr	Org. rulename	New rule	Org. rule
1	.rhosts	1	1
2	EXPLOIT STAT *	1	1
3	SITE EXEC	3 (3 rules)	3
4	serv-u dir. trav	2	2
5	CWD ~ attempt	1	3
6	.forward	1	1
7	CWD ~root	2	2

- False positives: 0

As we can see in Table 4, the new rules alert on the same traffic as the originals.

Statistics

Table 5: Differences between original and new rules

Ranking	**Occurences**	**Comment**
Rank 0	All	ref
Rank 1	0	
Rank 2	All	classtype (using priority directly)
Rank 3	0	
Rank 4	2	nocase not supposed to be included
Rank 5	7	pcre and distance missing
Rank 6	0	
Equal to original	0	Rank 0-2 not considered
Almost equal to original	3	Only difference is that our rules also include the FTP commands (RETR, SITE etc.). Rank 0-2 not considered

Table 5 shows that we have many rules missing fields with the rank of 5. The consequence of missing the `pcre`[1] field is not important since our procedure creates rules based on entire strings in the honeypot log file (exact content matches). We do however suffer from performance loss if several of our self generated rules could have been replaced by one rule using `pcre`. A missing distance results in lower performance, since `SNORT` would continue scanning the whole packet instead of stopping when the distance is achieved. The difference in number of `CWD ~ attempt` alerts in table 4 is because the original rule also alerts on the two occurences with `CWD ~root`.

Here is an excerpt from the bottom of the `SNORT` alert file, both the control and the experiment:

Experiment alerts when using the new rules

```
[**] [1:1000007:1] NEW FTP CWD ~root [**]
[Priority: 0]
04/14-15:34:41.376347 128.39.44.11:44798 -> 128.39.44.55:21
TCP TTL:64 TOS:0x0 ID:59088 IpLen:20 DgmLen:62 DF
***AP*** Seq: 0xEC1C4D5  Ack: 0xC5B02A19  Win: 0x5B4  TcpLen: 32
TCP Options (3) => NOP NOP TS: 1453204114 30225214
```

[1] If the exact format of the content we try to match is known, content should be used. For any variable data, use pcre.

Next are the alerts from the control experiment

```
[**] [1:336:10] FTP CWD ~root attempt [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
04/03-19:11:40.084219 128.39.44.11:60752 -> 128.39.44.55:21
TCP TTL:64 TOS:0x0 ID:12564 IpLen:20 DgmLen:62 DF
***AP*** Seq: 0x9A16AD01  Ack: 0xEFC0A422  Win: 0x5B4  TcpLen: 32
TCP Options (3) => NOP NOP TS: 515677826 30225254
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=1999-0082]
[Xref => http://www.whitehats.com/info/IDS318]

[**] [1:1672:11] FTP CWD ~ attempt [**]
[Classification: Detection of a Denial of Service Attack] [Priority: 2]
04/03-19:11:40.084219 128.39.44.11:60752 -> 128.39.44.55:21
TCP TTL:64 TOS:0x0 ID:12564 IpLen:20 DgmLen:62 DF
***AP*** Seq: 0x9A16AD01  Ack: 0xEFC0A422  Win: 0x5B4  TcpLen: 32
TCP Options (3) => NOP NOP TS: 515677826 30225254
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2001-0421][Xref =>
http://www.securityfocus.com/bid/9215][Xref =>
http://www.securityfocus.com/bid/2601]
```

As we see, both the original rule-set and the new rule-set alert on the same traffic when exactly the same traffic is replayed against the honeypot. The only difference is that the original rule set alerts twice on the `~root` attack, where as the new rule set only alerts once. The downside is that the original rule set has a general rule for attacks containing `CWD ~`, where as the new rule set has one rule for each different attack containing this string (one rule for `~root`, and one for `~nonexistinguser`. With a larger data set it would probably have been easier to find similarities in the attacks, and hence create more general rules. Another important difference is that the new rules always have a priority = 0. This is because we substitute the `classtype` option with `priority`, to get a special priority for all new rules. The reason for doing this is because we did not manage to extract the information needed to classify the attacks into classtypes. We then had three options, either use priority directly, do not include priority at all or make a new classtype for each service we create new rules for. (One classtype for zero-day web attacks and one for zero-day FTP attacks.) We chose the first option, because this gives the administrator an easy job if he wants to change the priority (if he finds the rule not so important).

Conclusion

In this experiment we were able to create `SNORT` rules based on the honeypot logs. These rules performed equally compared to the original rules used by `SNORT`. This only show that the honeypot logs sufficient information to re-create the rules, but not necessarily that our rules are just as good as the originals. If we were to create rules based only on the honeypot log, many new problems would occur. How to know what in the log file is an actual attack is one problem. This is not really an issue that will be considered in this thesis because by definition all traffic entering a honeypot is considered malicious. Therefore, the very usage of honeypots is a solution to this problem. But even though this is supposed to be the case, an attacker may very well use benign traffic in parts of his attack. Normal user may also connect to the honeypot by mistake. This would lead to a lot of false positives. So in the case of FTP attacks we conclude that the honeypot logs the information needed, but some kind of filtering must be used on the log file to remove known benign traffic and irrelevant strings like those listed below. This can easily be done by a script using the grep command, in conjunction with a known good database. It is important that the grep commands are specific enough so that they don't remove for instance all occurrences with "CWD", when "CWD ~" and CWD ..%20" is considered malicious traffic. The grep commands must therefore be applied with caution. In this

case, "CWD /" and "CWD" could be removed, but not any other occurrences like the before mentioned "CWD ~". By doing this, the probability of removing a new attack is greatly reduced, hence reducing the false negative rate. Another way of reducing benign traffic from the dataset is to use honeypot systems specifically developed to cope with the problem of getting benign traffic into the honeypot [45, 13].

We do not know what the result would have been if the amount of traffic had been larger. The only thing we proved here is that the honeypot logs the information needed to create the rules corresponding to the traffic produced by a Nessus FTP scan. Our rules also seem to be good enough to alert on the same attacks as the originals do. The major difference is that the original rules are more general, leading to less rules. They are also less susceptive to variations in the attacks because of the use of `pcre`.

6.6 WEB SERVER scan experiment

6.6.1 How the experiment is conducted

The virtual host on the honeypot, which gives the highest amount of different alerts, is used in this experiment (a SUsE host running an apache.sh script). This virtual host is then attacked by Nessus' web servers scan. Another machine running `SNORT` is checking the traffic produced by Nessus and the virtual host, and giving alerts on traffic it finds malicious.

6.6.2 Experiment 1 - Control

The following information is gathered from the log file where `honeyd` logs web attacks. `Honeyd` logged this information after a Nessus web server scan against one of the honeypots virtual hosts (128.39.44.50). In this particular case it was a SUsE host with an apache.sh script which was attacked. `SNORT` loads all default rule files except snmp.rules and icmp.rules. Below is a listing of the strings, which triggered alerts in the control experiment. We will consider all traffic not listed here but in the web.log file, as benign to make it more manageable. The actual log file consists of approximately 850 lines.

- GET /login.html HTTP/1.1
- GET /robots.txt HTTP/1.1
- GET /CVS/Entries HTTP/1.1
- GET /Admin_files/ HTTP/1.1
- GET /_vti_bin/ HTTP/1.1
- GET /admin_/ HTTP/1.1
- GET /backup/ HTTP/1.1
- GET /backups/ HTTP/1.1
- GET /cbi-bin/ HTTP/1.1
- GET /doc/ HTTP/1.1
- GET /iisadmin/ HTTP/1.1
- GET /iissamples/ HTTP/1.1
- GET /intranet/ HTTP/1.1
- GET /webalizer/ HTTP/1.1
- GET /_mem_bin/ HTTP/1.1
- GET /dms0/ HTTP/1.1
- GET /oprocmgr-status/ HTTP/1.1
- GET /server-info/ HTTP/1.1
- GET /server-status/ HTTP/1.1
- GET /srchadm/ HTTP/1.1

- GET /test-cgi/ HTTP/1.1
- GET /way-board/ HTTP/1.1
- GET /webcart/ HTTP/1.1
- GET /webcart-lite/ HTTP/1.1
- GET /www-sql/ HTTP/1.1
- GET /~root HTTP/1.1
- GET /DB4Web/ HTTP/1.1
- "PROPFIND / HTTP/1.1
- GET /mod_gzip_status HTTP/1.1
- "SEARCH / HTTP/1.1
- "GET / root HTTP/1.1
- GET /<script>cross_site_scripting.nasl</script> HTTP/1.1
- GET /NULL.ida HTTP/1.1
- GET /NULL.printer HTTP/1.1
- GET /_vti_bin/fpcount.exe HTTP/1.1
- GET /_vti_bin/shtml.dll/_vti_rpc HTTP/1.1
- GET /iisadmpwd/aexp.htr HTTP/1.1
- GET /iisadmpwd/aexp2.htr HTTP/1.1
- GET /iisadmpwd/aexp2b.htr HTTP/1.1
- GET /iisadmpwd/aexp3.htr HTTP/1.1
- GET /iisadmpwd/aexp4.htr HTTP/1.1
- GET /iisadmpwd/aexp4b.htr HTTP/1.1
- GET /search?NS-query-pat=../../../../../../../../../etc/passwd HTTP /1.1
- GET /_vti_inf.html HTTP/1.1
- GET /a.jsp/<SCRIPT>alert(document.domain)</SCRIPT> HTTP/1.1
- GET /cgi-bin/test.cgi.bat? | echo HTTP/1.1
- GET ../../../../../../etc/passwd HTTP/1.1
- GET /../../../../../../../../../etc/passwd HTTP/1.1
- GET //../../../../../../../../../etc/passwd HTTP/1.1
- GET /%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd HTTP/1.1
- GET /./../../../../../../../../../etc/passwd HTTP/1.1

The original log file has been manipulated to only include strings which are important to generate rules out of. This manipulation has been done by using `grep`[2]. From the example below, we want to keep the string containing `"GET /login.html`. This is done by removing the `-MARK-` string and all strings below `"GET`. The options used with `grep` are -w for words and -v if we want to keep everything except the lines containing the pattern we search for. In the case of the example (Fig. 8) we could use the following command: **less web.log | grep GET > tmp**. This would give us all lines containing `GET`. But since there are attacks that do not use a GET line, we had to manipulate the log file the hard way by removing lines without useful information with `grep -w -v <a word>`.

6.6.3 Experiment 2 - New rules

This experiment is conducted in the same way as the control, but with the new rules loaded instead of the original web rules. The same traffic is sent towards the virtual

[2]General Regular Expression Parser. Grep is a UNIX program that looks for patterns found in files and reports on their occurrences.

```
--MARK--,"Tue Apr  4 12:46:01 CEST 2006","apache/HTTP","128.39.44.11",
"128.39.44.50",44797,80,
"GET /login.html HTTP/1.1
Connection: Close
Host: 128.39.44.50
Pragma: no-cache
User-Agent: Mozilla/4.75 [en] (X11, U; Nessus)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
",
--ENDMARK--
```

Figure 8: How a web attack looks like in the honeypot's log file `web.log`

host (not only the strings listed above). We still consider only the strings from above to be malicious traffic, and all other traffic as benign. This is against the definition of traffic towards a honeypot, but makes it easier to measure the difference in performance (meaning number of alerts). We use this approach because we need to be able to compare the original rules with the ones we create.

WEBSERVER-scan against a virtual SUsE machine with an apache.sh script

In this section we try to create SNORT rules based on the strings from section 6.6.2. The goal is to see if we are able to create useful rules based on that information. We will compare the new rules to the originals by measuring the differences based on the experiment results found in the next section, in addition to the ranking of each missing field. All the rules are made by hand following the procedure presented below, bearing in mind that only information from the honeypot may be used. Since all the attacks against the web server consist of "some string", we will use "content" and "uricontent" as the triggering field in our rules. The difference between the two is that "uricontent" only checks the URI[3] of an HTTP packet as opposed to the whole packet as "content" does. One major difference between the experiment and what would have been done in the "wild", is that there would have been created rules for all the strings in the web.log file. This is because of the fact that all traffic towards a honeypot is considered malicious.

Procedure for creating web rules

The following is a sketch of the web rules creating procedure used in this thesis:

```
int $count = 1000000;  \\ used for sid numbering
int $X = 5;            \\ used for LCS minimum lenght
string $cont, $cont2;  \\ storing content (signature)
string $cont_OR_uri;   \\ use content or URIcontent
int $one_OR_two;       \\ one or two contents switch

Use "grep" to remove irrelevant data in the log file.
Sort the file alphabetically.
if many lines starting with equal content (3 or more), use LCS to
find what is equal. (Minimum result from LCS is X characters).
$one_OR_two = 1;

while (file not empty){
if !(../ || %2e%2e || ./ in logfile line) {
$cont_OR_uri=uricontent:;
```

[3]Uniform Resource Identifier. SNORT runs the URI trough the http_inspect pre-processor to normalize it. The pre-processor fix URI's that have been mangled in an attempt to evade the IDS. When using "uricontent" only the actual content of the URI is considered, not http:\\ ,GET and so on.

```
$cont="string before HTTP/1.1" (Not include GET if present)
}

else
$cont_OR_uri=content:;
if (string before and after ../ or %2e%2e or ./) {
$cont="string before first ../ or %2e%2e or ./
$cont2="string after last ../ or %2e%2e or ./ and HTTP/1.1"
$one_OR_two=2
}
else
$cont="string after last ../ or %2e%2e or ./"

if (one_OR_two = 1) {
print to rulefile {
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
$cont + $cont2"; flow:to_server,established; $contORuri:$cont;
nocase; priority:0; sid:$count; rev:1;)
}

elseif (one_OR_two = 2) {
print to rulefile {
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
$cont"; flow:to_server,established; $contORuri:$cont; nocase;
priority:0; $conORuri:$cont2; nocase; sid:$count; rev:1;)
}
else
ERROR
$count ++;
}
```

This procedure is an attempt to automate the creating of the rules. Most fields in the print section are static because they always appear in the way they do here in the corresponding originals. The information for `flow` and `protocol` is available in the honeypot file `honeyd.log` (see section 6.7). We also use `nocase` in all of our rules to make sure `SNORT` ignore the case of the letters. The procedure is created based on knowledge from the control experiment to re-created known rules, and hence not valid if deployed in the wild. We still create rules only based on the honeypots log files, but a little knowledge of the originals, like the `flow`, is statically implemented.

Below are three examples of rules made by hand from the honeypot log. These rules have been made following the procedure above. For a complete listing of rules please see Appendix B.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
/login.htm"; flow:to_server,established; uricontent:"/login.htm"; nocase;
priority:0; sid:1000010; rev:1;)
```

Different from the original: [1564]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
/robots.txt"; flow:to_server,established; uricontent:"/robots.txt";
nocase; priority:0; sid:1000011; rev:1;)
```

Different from the original: [1852]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
/CVS/Entries"; flow:to_server,established; uricontent:"/CVS/Entries";
nocase; priority:0; sid:1000012; rev:1;)
```

Different from the original: [1551]

- no reference
- no classtype:web-application-activity

Results from testing the new rules compared to the originals

In this experiment we have added the new rules into SNORT. Then we have done exactly the same as we did in the control experiment.

SID	Rulename	New rule	Org. rule
1564	login.htm access	1	1
1852	robots.txt access	1	1
1551	/CVS/Entries access	1	1
1212	Admin_files access	1	1
1288	/_vti_bin/ access	3	3
1385	mod-plsql admin access	1	1
1213	backup access (two rules)	2	2
1668	/cgi-bin/ access	12	1
1560	/doc/ access	1	1
993	iisadmin access	1	1
1402	iissamples access	1	1
1214	intranet access	1	1
1847	webalizer access	1	1
1286	_mem_bin	1	1
1872	Oracle Dynamic Monitoring Services dms access	1	1
1874	Oracle Java Process Manager access	1	1
1520	server-info access	1	1
1521	server-status access	1	1
1040	srchadm access	1	1
835	test-cgi access	1	2
896	way-board access	1	1
1125	webcart access	1	1
1848	webcart-lite access	1	1
887	www-sql access	1	1
1145	/~root access (two rules)	2	2
2060	DB4Web access	1	1
1079	WebDAV propfind access	1	1

Continued on next page

SID	Rulename	New rule	Org. rule
974	Directory traversal attempt	No	6
1112	http directory traversal	No	7
2156	mod_gzip_status access	1	1
2091	WEBDAV nessus safescan attempt	No	1
1070	WebDAV search access	1	1
1497	cross site scripting attempt (two rules)	2	2
1242	.ida access	1	1
971	.printer access	1	1
1013	fpcount access	1	1
940	shtml.dll	1	1
937	_vti_rpc (included in another rule)	No	1
1018	iisadmpwd attempt	6	6
987	.htr access	6	6
1487	/iisadmpwd/aexp2.htr access (included in another rule)	No	1
1828	iPlanet Search directory traversal attempt	1	1
1122	/etc/passwd	6	6
990	_vti_inf.html access	1	1
976	.bat? access (included in another rule, but does not work)	No	1

Table 6: Number of alerts from each rule when a Nessus web servers scan is run

As Table 6 shows, the new rules perform pretty well. The major difference is that the new rules usually alert on the complete string in which the attack is included. This makes the new rules less specific then the originals, but they do alert, which is the most important part. One must bear in mind though that this experiment uses static traffic, meaning that every time it is run it produces the same traffic. Because of this it is difficult to measure the false negative and false positive rate expected when run in the wild.

What traffic caused different alerts?

By using the Unix command `diff`[4], we get the following differences when comparing which strings the new rules alerted on comparing to the originals.

```
> "GET /cgi-bin/zQbVo4pJ9OXq.html HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.cgi HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.sh HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.pl HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.inc HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.shtml HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.php HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.php3 HTTP/1.1
> "GET /cgi-bin/zQbVo4pJ9OXq.cfm HTTP/1.1
> "GET /cgi-bin/com5.pl HTTP/1.1
< "GET /cgi-bin/test-cgi.bat?|echo HTTP/1.1 (/cgi-bin triggers this rule,
                                              but it should be /test-cgi)
```

[4]Diff is a file comparison utility that outputs the differences between two files.

All lines with ">" is traffic the new rules alerted on but not the originals. (False positives). All lines with "<" is traffic the new rules should have alerted on (False negatives).

Statistics

Table 7: Differences between original and new rules

Ranking	**Occurences**	**Comment**
Rank 0	All	ref
Rank 1	0	
Rank 2	All	classtype missing, using priority instead
Rank 3	0	
Rank 4	3	within, nocase and depth:8 are missing
Rank 5	1	pcre is missing
Rank 6	3	content, uricontent
No rule	4	sid:974,1112,2091,976,(937,1487). 937 and 1487 are included in other rules
Equal to original	20	Rank 0-2 not considered
Almost equal to original	9	Only difference is a / to much in the uricontent or content field. Rank 0-2 not considered

Table 7 shows that we are having more problems re-creating rules in this experiment than we had in the FTP experiment. We have fewer occurrences with rank 5, but here we have 3 occurrences with the rank of 6. The occurrences with rank 6 are re-created rules missing one of two content elements. Because of the simple procedure we use for extracting content matching strings, we manage to make the rule work with only one of the two content elements. This makes our rules more specific until we have enough almost similar attacks to extract similarities from. A reason for us to have more problems re-creating rules for web attacks is probably because we have about 6 times as many attacks in this experiment. The information needed to fix the missing fields in rank 6 is available in the honeypot log, but our simple procedure is not able to extract this information. The information needed to re-create the rules we did not manage to re-create was not found in the `web.log` file used in this experiment. It is possible that this information can be found in `honeyd.log` or by using `tcpdump` on all traffic in and out from the honeypot. Alert statistics not including rules we did not manage to re-create at all (Rows in table 6 with "No"):

- False negatives: 1/51 = 1,961% (Because the rule did not work. Really 0 because this string is alerted on by another rule)
- False positives: 11 (really 10 because one of the false positives are actually an attack supposed to be alerted on by a rule we did not manage to re-create)

Alert statistics overall:

- False negatives: 15 (Mostly because we did not manage to re-create some of the rules)
- False positives: 11 (really 10 because one of the false positives is actually an attack supposed to be alerted on by a rule we did not manage to re-create)

If we consider each line from the list found in 6.6.2 as attacks (51 attacks), and only demand that an alert for each line is present (regardles of what it actually alert on), we get the following TPR/FPR for the new rules:

$$TPR = \frac{51}{51 + 0} = 1$$

$$FPR = \frac{10}{10 + 773} = 0,128$$

The original rules are used as a reference here, hence the TPR = 1 and the FPR = 0. This does not mean that the original rules perform perfectly, but we consider the original rules as perfect in order to have something to compare with. As we can see from the calculation above, the new rules have a TPR = 1, and a FPR = 0,128. This shows that the new rules alert on all lines found in 6.6.2, but also alert on some lines not considered as attacks. Since this system is supposed to detect zero-day attacks, we believe that a few False Positives is a good tradeoff to get a TPR = 1.

If we consider all traffic collected by the honeypot as malicious (834 attacks, as the definition of a honeypot states), we get the following TPR and FPR:

Original rules:

$$TPR = \frac{51}{51 + 783} = 0,06115$$

$$FPR = \frac{0}{0 + 0} = 0$$

New rules:

$$TPR = \frac{61}{61 + 773} = 0,073$$

$$FPR = \frac{0}{0 + 0} = 0$$

These results show that our rules actually perform better than the originals. The TPR for our new rules would be considerable higher if we had carried out a full-scale experiment. It is important to note that we only tried to make rules for attacks `SNORT` alerted on in the control experiment. We claim that it is possible to create rules for the Nessus web servers scan that would give a TPR $\approx$ 1, based only on data collected by the honeypot.

What caused the differences?
To be certain Nessus sends the same traffic each time, we used the Linux command `diff` to find differences between the first and second time the experiment was run. To see the full output from this test, see Appendix C. As this shows, the differences between the first and second attempt is irrelevant for the experiment.

The new rules, which performed differently from the control are:

- backup
- cgi-bin access
- test-cgi access
- ~root access
- cross site scripting attempt
- + the rules which were not re-created

Number one in table 8 is a result of the procedure failing to see that /backup/ and /backups/ could be in one rule. The result is that two rules are created, only resulting in

Table 8: Differences between original and new rules

Nr	Rulename	Org. rule	New rule
1	/backup	uricontent:"/backup"	uricontent:"/backup/"
2	/cgi-bin	uricontent:"/cgi-bin/ + content:"/cgi-bin/ HTTP"	uricontent:"/cgi-bin/"
3	/test-cgi	uricontent:"/test-cgi"	"uricontent:"/test-cgi/"
4	/~root	"uricontent:/~root"	"uricontent:"/~root/"
5	cross site scripting	content:"<SCRIPT>"	uricontent:"/<script>cross_site_scripting.nasl</script>"

a microscopic performance loss in SNORT.
Result: 0 False Positives/Negatives, but performance loss in loading one extra rule into SNORT.

The reason number two in table 8 differed so much regarding number of alerts is because of the missing *content:"/cgi-bin/ HTTP*". The original rule only alerts on occurrences with /cgi-bin/ alone, as opposed to the new one which alerts on all occurrences with /cgi-bin/. This leads to 11 false positives when we consider all traffic which was not alerted on in the control experiment as benign.
Result: 11 False Positives.

The reason number three in table 8 only caught one of the two lines with /test-cgi, is because the new rule only searches for /test-cgi/ which don't alert on /test-cgi.bat? as the original do. The procedure created a rule to search for the string containing /test-cgi.bat?, but it did not work with SNORT because this particular string included a pipe symbol making SNORT believe the content after the pipe symbol is in a hexadecimal format. (This string is alerted on by another rule, but this is only a coincidence)
Result: 1 (0) False Negative.

The reason number four differ is for the same reason as for number two. It has a / in the end of the trigger string, resulting in a false negative. The happy side is that the procedure has created a rule for the string this rule failed to alert on, actually making this rule redundant.
Result: 0 False Positives/Negatives

The reason number five does not perform equally as the original is because of the procedure's lack of understanding what in the strings is important. The string producing the false negative in this case is captured by another rule, making the procedure actually covering both lines containing <script>.
Result: 0 False Positives/Negatives, but performance loss in loading one extra rule into SNORT.

Alert when using the new rules

The following alert is produced by SNORT when loaded with the new rules we have crated. This rule alerts on traffic matching "/_vti_bin/shtml.sll/_vti_rpc".

```
[**] [1:1000044:1] NEW WEB /_vti_bin/shtml.dll/_vti_rpc [**]
[Priority: 0]
04/11-13:52:40.269727 128.39.44.11:43740 -> 128.39.44.50:80
TCP TTL:64 TOS:0x0 ID:28552 IpLen:20 DgmLen:324 DF
***AP*** Seq: 0xD4FF684B  Ack: 0x7AEBFC2F  Win: 0x16D0  TcpLen: 20
```

Alerts when using the original rules

These alerts are produced by SNORT when loaded with the original rules. The three alerts below are triggered on the same string as above. As this shows, the original rules divide this string into three different alerts. This results in a more specific message to the administrator with references to each of the vulnerabilities, but also gives him more alerts to read through.

```
[**] [1:940:15] WEB-FRONTPAGE shtml.dll access [**]
[Classification: access to a potentially vulnerable web application] [Priority: 2]
04/04-12:50:56.906906 128.39.44.11:45582 -> 128.39.44.50:80
TCP TTL:64 TOS:0x0 ID:29615 IpLen:20 DgmLen:324 DF
***AP*** Seq: 0x3A8FE212  Ack: 0x2C32288D  Win: 0x16D0  TcpLen: 20
[Xref => http://www.microsoft.com/technet/security/bulletin/ms00-060.mspx]
[Xref => http://cgi.nessus.org/plugins/dump.php3?id=11395]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0746]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0413]
[Xref => http://www.securityfocus.com/bid/1595]
[Xref => http://www.securityfocus.com/bid/1594]
[Xref => http://www.securityfocus.com/bid/1174]
[Xref => http://www.whitehats.com/info/IDS292]

[**] [1:1288:8] WEB-FRONTPAGE /_vti_bin/ access [**]
[Classification: access to a potentially vulnerable web application] [Priority: 2]
04/04-12:50:56.906906 128.39.44.11:45582 -> 128.39.44.50:80
TCP TTL:64 TOS:0x0 ID:29615 IpLen:20 DgmLen:324 DF
***AP*** Seq: 0x3A8FE212  Ack: 0x2C32288D  Win: 0x16D0  TcpLen: 20
[Xref => http://cgi.nessus.org/plugins/dump.php3?id=11032]

[**] [1:937:10] WEB-FRONTPAGE _vti_rpc access [**]
[Classification: access to a potentially vulnerable web application] [Priority: 2]
04/04-12:50:56.906906 128.39.44.11:45582 -> 128.39.44.50:80
TCP TTL:64 TOS:0x0 ID:29615 IpLen:20 DgmLen:324 DF
***AP*** Seq: 0x3A8FE212  Ack: 0x2C32288D  Win: 0x16D0  TcpLen: 20
[Xref => http://cgi.nessus.org/plugins/dump.php3?id=10585]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2001-0096]
[Xref => http://www.securityfocus.com/bid/2144]
```

Conclusion

For the sake of this experiment our procedure created fewer rules, but still alerted on all but one of the strings containing possible malicious content. This proves again that honeyd logs are suitable for creating rules out of. How the honeyd logs would look like when deployed in the wild is not considered here, but we claim that the logging capabilities are sufficient. Since we only used traffic produced by Nessus it is hard for us to say anything about how the rule generating procedure would perform when tested on live/wild traffic. The only thing we know for sure is that honeyd logs enough information to make meaningful rules from, but that extracting the signature itself may be a problem (especially for polymorphic worms). Suggestions for improvements are documented in the future work chapter.

6.7 Extracting `flow` and `protocol`

The rule option flow and rule header field protocol is possible to extract from honeyd's honeyd.log file. A working procedure for doing this is not explained here, but a possible

approach is suggested. The file `honeyd.log` logs all connections to the honeypot (how to interpret information in this file is given in subsection 5.2.4. Below is an excerpt from the file:

```
Date-11:39:45.2204 tcp(6) S xxx.xxx.xxx.132 21133 128.39.xxx.47 80 [Win XP SP1]
Date-11:39:45.5088 tcp(6) E xxx.xxx.xxx.132 21133 128.39.xxx.47 80: 0 1058
```

This information could then be linked up to other log files to be able to find flow and protocol for a specific attack. From the information above we could extract the following:

```
Flow: from: xxx.xxx.xxx.132 21133 to 128.39.xxx.47 80
            Results in a: flow:to_server,establieshed
Protocol: tcp
$HOME_NET and $HTTP_PORTS (by checking destination address and portnumber
                           against the snort.conf file)
```

7 General conclusion and future work

In this thesis we have defined a new IDS rule generating procedure based on information obtained from honeypot logs made by `honeyd`. Several experiments were performed in order to see if the honeypot was able to log sufficient information to create rules from. As proven in the experiments, this seems to be the case since we were able to re-create nearly all the rules we aimed at re-creating. The test results show that the rule generating procedure gave a few false positives and none false negatives. The main goal of the experiments was to see if the honeypot logged enough information to make rules out of. To find the answer to this we created a system for ranking each rule option field in the `SNORT` rule. As the results from this ranking system show, the honeypot logs the most important fields necessary for rule generating. This confirms that `honeyd` is suitable for generating `SNORT` rules. Our rule generating system was made to work with traffic generated by Nessus, and hence would not work perfectly in the wild. Polymorphic worms for instance would make our rule generating system create rules which would be useless since the same signature would not appear twice. If we compare our generating method to other signature generating systems [26, 39], we can state that our method includes neccessary rule options these systems does not (i.e. `flow`, `priority`. Also our rules, because of the fact that we make the rules with more options, would perform faster when implemented into `SNORT`. One example is the `flow` option, which make `SNORT` only check traffic matching the specified flow. Also the use of names on different services, i.e. HTTP_SERVERS, makes `SNORT` perform faster.

We now return to the research questions to summarize the findings:

1. Is `honeyd` suitable for detecting new attacks?

   This question is answered by the experiments, which give a conclusion that it is suitable for this. We actually did not detect any new attacks, this due to the desire to make the experiments reproducible, hence using Nessus as a traffic generator. If the honeypot logs sufficient data about Nessus scans, it most likely would log enough data about other attacks too.

2. How to translate data captured by `honeyd` into `SNORT` rules?

   To answer this question a procedure for converting honeypot log files into `SNORT` rules was made. This procedure has a few limitations, but in the setting we made it work for, it functions in such a manner that it produces a low false positive and false negative ratio.

3. How good rules can be made based on data logged by `honeyd`?

   We created a measurement system to rank each option field used by the original `SNORT` rules. The option fields were given a rank based on the impact they have on the rules triggering capabilities. Ranking was given by a number from 0-6, where 6 is given to the most important option needed to make the rule trigger. As our experiments demonstrate, the rules we re-created had a low number of missing options that were important to make them trigger. This confirms that `honeyd` is suitable for collecting attack data to make IDS rules out of.

7.1 Future work

Based on the findings in this thesis, we suggest the following point marks as future work:

- Improve the rule generating procedure to work with live/wild traffic.
- Create a metric for measuring the manual workload needed to fix rules with missing option fields and non good option fields. If this metric were to be acknowledged internationally, it would be easier to create good rulegenerating systems when there is a "standard" to fulfill.
- Implement the improved system for data collection (Section 5.7) to see if this gives a better dataset to create rules from.
- Test known rulegenerating systems with Nessus traffic, and compare the generated rules to the ones presented in this thesis.
- Implement a known procedure for extraction of polymorphic worm signatures (Polygraph/Nemean) or make one.
- Use a different traffic generator than Nessus to get a different data set.
- Record live traffic in the honeypot with i.e. `tcpdump`, generate rules and replay the traffic with the new rules loaded into `SNORT` to see if the rules are good enough. Expected result is that the `msg` option field might not be as descriptive as the ones in our experiments. This due to the fact that live traffic contains several attack types not discussed in this thesis (i.e. worms, which `SNORT` alerts on based on hexadecimal strings).

Bibliography

[1] The honeynet project. http://www.honeynet.org/. (Visited Oct. 2005).

[2] Honeynets article. http://project.honeynet.org/papers/honeynet/index.html. Visited Oct.

[3] Metasploit. http://www.metasploit.com/index.html.

[4] Nessus, vulnerability scanner. www.nessus.org.

[5] Personopplysningsloven. http://www.lovdata.no/all/nl-20000414-031.html.

[6] Saint. http://www.saintcorporation.com/products/vulnerability_scan/saint/saint_scanner.html.

[7] Sebek. http://www.honeynet.org/tools/sebek/. (Visited Nov. 2005).

[8] Snort. http://www.snort.org/. (Visited Nov. 2005).

[9] Specter, intrusion detection system. www.specter.com. (Visited Dec. 2005).

[10] Vlad. http://www.bindview.com/Services/RAZOR/Utilities/Unix_Linux/vlad.cfm.

[11] Wikipedia, the free encyclopediaa. http://en.wikipedia.org.

[12] Rfc 2828. http://rfc.net/rfc2828.html, 2000.

[13] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, , and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *In Proceedings of the 14th Usenix Security Symposium*, 2005.

[14] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, Fort Washington, 1980.

[15] Mick Bauer. Paranoid penguin: checking your work with scanners, part ii: Nessus. *Linux J.*, 2001(86):11, 2001.

[16] Reto Baumann. Honeyd - a low involvement honeypot in action. *Originall published as part of the GCIA practical*, page 14, 2003.

[17] E. Biermann, E.Cloete, and L.M. Venter. A comparison of intrusion detection systems. *Computers & Security*, 20:676–683, 2001.

[18] Raouf Boutaba, Kevin Almeroth, Ramon Puigjaner, Sherman Shen, and James P. Black. Efficient deployment of honeynets for statistical and forensic analysis of attacks from the internet. *Lecture Notes in Computer Science*, 34622005:756–767, 2005.

[19] A. Broder. Some applications of rabin's fingerprinting method, 1993.

[20] Roshen Chandran and Sangita Pakala. Simulating networks with honeyd. Technical report, Dec 2003.

[21] Mark Cooper, Stephen Northcutt, Matt Fearnow, and Karen Frederick. *Intrusion Signatures and Analysis*. SAMS, 2001.

[22] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[23] Van Jacobson, Craig Leres, and Steven McCanne. Tcpdump, intercept and display communications. www.tcpdump.org. (Visited Dec. 2005).

[24] Hyang-Ah Kim and Brad Karp. Autograph, toward automated, distributed worm signature detection. *USENIX Security Symposium*, pages 271–286, 2004.

[25] Jack Koziol. *Intrusion detection with SNORT*. SAMS, 2003.

[26] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34:51–56, 2004.

[27] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222. ACM Press, 2005.

[28] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.

[29] Peter Mell, Vincent Hu, Richard Lippmann, Josh Haines, and Marc Zissman. An overview of issues in testing intrusion detection systems. http://csrc.nist.gov/publications/nistir/nistir-7007.pdf. NIST IR 7007.

[30] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *Network, IEEE*, Volume 8:26 – 41, 1994.

[31] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. *IEEE Computer Society Washington, DC, USA*, 2005.

[32] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[33] Fabien Pouget and Thorsten Holz. A pointillist approach for comparing honeypots. *Lecture Notes in Computer Science*, 3548:51, 2005.

[34] Niels Provos. Honeyd, virtual honeypot. http://www.honeyd.org/. (Visited Dec. 2005).

[35] Niels Provos. A virtual honeypot framework. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, August 2004.

[36] Shai Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of nids attacks. *20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 28–38, 2004.

[37] Karthik Sadasivam, Banuprasad Samudrala, and T. Andrew Yang. Design of network security projects using honeypots. *J. Comput. Small Coll.*, 20(4):282–293, 2005.

[38] David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, 1983.

[39] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. *USENIX Security Symposium*, pages 45–60, 2004.

[40] L. Spitzner. *Know Your Enemy : Learning about Security Threats*. Addison-Wesley Professional, 2004.

[41] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, 2002.

[42] Lance Spitzner. Honeytokens: The other honeypot, 2003. www.securityfocus.com/infocus/1713.

[43] Lance Spitzner. Problems and challenges with honeypots. http://www.securityfocus.com/infocus/1757, 2004.

[44] A. Sundaram. An introduction to intrusion detection. *The ACM student magazine*, 1996.

[45] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *In Proceedings of the IEEE Infocom 2005*, 2005.

[46] Urjita Thakar, Sudarshan Varma, and A.K. Ramani. Honeyanalyzer : Analysis and extraction of intrusion detection patterns & signatures using honeypot.

[47] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *RAID*, pages 227–246, 2005.

[48] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, pages 203–222, 2004.

[49] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. *14th USENIX Security Symposium*, pages 97–112, 2005.

A Installation and configuring

A.1 Installing honeyd-1.0a-rc2.tar.gz on Fedora Core 3

1. Install Fedora Core 3 with all necessary developement packages.
2. Update *yum.conf* with the following:
   ```
   [dries]
   name=Extra Fedora rpms dries - $releasever - $basearch
   baseurl=http://ftp.belnet.be/packages/dries.ulyssis.org/fedora/linux/$releasever/
   $basearch/dries/RPMS/
   ```
3. Run this command in a bash shell: *yum update*
4. Install libraries `Honeyd` needs: *yum install libdnet.i386* && *arpd.i386* (`arpd` should be avoided in production networks, use proxy arp instead)
5. Download *libevent1.0e* and *libevent-devel1.0e* from *http://rpmfiend.net* and install them
6. Extract *honeyd-1.0a-rc2.tar.gz*
7. Go to the folder containing *honeyd-1.0a-rc2* files (in a bash shell)
8. Run the following commands:
   ```
   # ./configure - -without-python
   # make
   # make install
   ```

`Honeyd` files are now placed in the */usr/local/share/honeyd/* folder. Before you start `Honeyd` you need to configure the *.conf* file in */usr/local/share/honeyd/* We have made the .conf file based on the IP range *128.39.44.32/27* (*128.39.44.32-128.39.44.63*). This range has not been subnetted in the router, and therefore we use the entire range of the subnet as hosts. Hence the first and last address in the range should be the broadcast (.63) and network address (.32). Because of problems encountered when not binding all IP addresses in the range to specific hosts (segmentation fault), we have used the bind command also on the default template. This is our .conf file as of April the 10th 2006: */usr/local/share/honeyd/honeyd.conf*

```
################################################
#Honeyd configuration file made by Vidar Grønland#
################################################

route entry 128.39.44.32 network 128.39.44.32/27
route 128.39.44.32 link 128.39.44.32/27

#############################################################################################

# WINDOWS XP WORKSTATION
create windows
set windows personality "Microsoft Windows XP Professional SP1"
set windows uptime 1728650
add windows tcp port 80 "sh /scripts/win32/web.sh"
set windows default tcp action reset
set windows default udp action reset
set windows ethernet "3Com"

#Debian-specific (use nobody = 65534 instead of 32767)
#set windows uid 65534 gid 65534

#############################################################################################
```

```
# WINDOWS 2003 SERVER
create win_serv
set win_serv personality "Microsoft Windows Server 2003 Enterprise Edition"
set win_serv uptime 8728650
add win_serv tcp port 21 "sh /scripts/win32/win2k/msftp.sh $ipsrc $sport $ipdst $dport"
add win_serv tcp port 80 "sh /scripts/win32/win2k/iis.sh $ipsrc $sport $ipdst $dport"
set win_serv default tcp action reset
set win_serv default udp action reset
set win_serv ethernet "3Com"

###########################################################################################
# LINUX HOST
create linux
set linux personality "Linux kernel 2.4.20"
set linux uptime 15111242
add linux tcp port 21 "sh /scripts/unix/linux/suse7.0/wuftpd.sh $ipsrc $sport $ipdst $dport"
add linux tcp port 80 "sh /scripts/unix/linux/suse8.0/apache.sh $ipsrc $sport $ipdst $dport"
set linux default tcp action reset
set linux default udp action reset
set linux ethernet "3Com"
#Debian-specific (use nobody = 65534 instead of 32767)
#set linux uid 65534 gid 65534

###########################################################################################
# LINUX SuSE HOST
create linux_SuSE
set linux_SuSE personality "Linux 2.4.7 (X86)"
#set linux_SuSE personality "Linux kernel 2.2.13 (SuSE; X86)"
set linux_SuSE uptime 8111242
add linux_SuSE tcp port 21 "sh /scripts/unix/linux/suse8.0/proftpd.sh $ipsrc $sport $ipdst $dport"
add linux_SuSE tcp port 80 "sh /scripts/unix/linux/suse8.0/apache.sh $ipsrc $sport $ipdst $dport"
set linux_SuSE default tcp action reset
set linux_SuSE default udp action reset
set linux_SuSE ethernet "3Com"

###########################################################################################
create default
set default personality "Microsoft Windows XP Home Edition"
add default udp port 135 open
add default tcp port 137 open
add default udp port 137 open
add default tcp port 139 open
set default default tcp action reset
set default default udp action block
set default default icmp action open
set default uptime 1523472
#set default uid 17 gid 17
set default ethernet "3Com"

###########################################################################################
# CISCO ROUTER
create router
set router personality "Cisco 7206 running IOS 11.1(24)"
set router default tcp action reset
set router default udp action reset
add router tcp port 22 "sh /scripts/unix/linux/suse8.0/ssh.sh $ipsrc $sport $ipdst $dport"
add router tcp port 23 "perl /scripts/router/router-telnet.pl"
add router tcp port 80 open
set router uid 65000 gid 32767
set router uptime 57327950

###########################################################################################
bind 128.39.44.32 router

bind 128.39.44.33 win_serv
bind 128.39.44.34 win_serv
```

```
bind 128.39.44.35 windows
bind 128.39.44.36 windows
bind 128.39.44.37 windows
bind 128.39.44.38 windows
bind 128.39.44.39 windows
bind 128.39.44.40 windows
bind 128.39.44.41 windows
bind 128.39.44.42 windows
bind 128.39.44.43 windows
bind 128.39.44.44 windows
bind 128.39.44.45 windows
bind 128.39.44.46 windows
bind 128.39.44.47 windows
bind 128.39.44.48 windows
bind 128.39.44.49 windows

bind 128.39.44.50 linux_SuSE
bind 128.39.44.51 linux
bind 128.39.44.52 linux
bind 128.39.44.53 linux
bind 128.39.44.54 linux
bind 128.39.44.55 linux
bind 128.39.44.56 linux
bind 128.39.44.57 linux
bind 128.39.44.58 linux
bind 128.39.44.59 linux

bind 128.39.44.60 default
bind 128.39.44.61 default
bind 128.39.44.62 default
bind 128.39.44.63 default
```

All the scripts that simulate services must be in the folders specified in the .conf file, it is also vital that all scripts and files used by `Honeyd` is owned by the correct user. We run `Honeyd` by the user **Honeyd**, which can be created by using the *useradd* command from */usr/sbin/*:

```
# ./useradd -m -s /sbin/nologin -d /usr/local/share/honeyd -u 65000 honeyd
```

Now, to start `Honeyd` you first need to start the arp daemon (if you don't use proxy arp). This is done from the folder */usr/sbin/*:

```
# ./arpd -d 128.39.44.32/27
```

If you want to use proxy arp, then repeat this for all IP's:

```
# arp -s 128.39.44.32 <honeyd's mac adr> permanent pub
```

Then open a new shell and do the following:
SU to root. Goto */usr/local/share/honeyd/* and create a file named *honeyd.log*. Change the owner of this file to the **Honeyd** user (*chown honeyd **).

To start `honeyd`, type in the following into a shell standing in *user/local/share/honeyd*:

```
# honeyd -d -u 65000 -f honeyd.conf -x xprobe2.conf -p nmap.prints -a
nmap.assoc -l honeyd.log 128.39.44.32/27
```

- `-d` tells `Honeyd` not to daemonize it self. This makes `Honeyd` show all traffic directly on to the screen

- `-u` tells `Honeyd` to run using the **Honeyd** user
- `-f` is the name of the configuration file
- `-x, -p` and `-a` are files used by `honeyd` for fooling nmap and xprobe scans
- `-l` is the name of the log file
- `128.39.44.32/27` tells `honeyd` to only listen to traffic directed to this segment of adresses

To stop `honeyd`, use ctrl+c. If you are to run `honeyd` for a longer period of time, it is advised to run it in a sandbox environment using `systrace` or `chroot`. This prevents an attacker from exploiting bugs in your `honeyd` scripts.

Depending on which scripts you use, you also may have to create files named *web.log* and *honeyd.txt*.
We have made a simple perl script to delete and create the logging files needed:

```
#!/usr/bin/perl

print ‘rm web.log‘;
print ‘rm iis.log‘;
print ‘rm honeyd.log‘;
print ‘rm honeyd.txt‘;
print ‘rm web.sh.log‘;

print ‘echo  >> /usr/local/share/honeyd/web.log‘;
print ‘echo  >> /usr/local/share/honeyd/iis.log‘;
print ‘echo  >> /usr/local/share/honeyd/honeyd.log‘;
print ‘echo  >> /usr/local/share/honeyd/honeyd.txt‘;
print ‘echo  >> /usr/local/share/honeyd/web.sh.log‘;
print ‘chown honeyd *‘;
```

A.2 Problems during installation of various programs

During the installation of `honeyd`, we encountered a few problems. We had estimated to use about to-three days to make `honeyd` work. This was a bad estimate, since we actually used about 15 days to make it work properly. Because of limited experience with Unix based operating systems, solving the different problems were difficult. We also posted questions on the *honeyd.org* forum, hoping to get solutions to our problems. As it turned out, no one had answers to our specific problems. But after trying `honeyd` on Ubuntu, Fedora Core 4 and Fedora Core 3 it finally worked after some tweaking on the Fedora Core 3 distribution. The problems encountered are described below.

A.2.1 Installing `honeyd` on Ubuntu 5.10

The *www.honeyd.org* webpage states that `honeyd` requires `libevent, libdnet` and `libpcap` to be able to compile. On our first attemt to compile `honeyd` on a Ubuntu machine we encountered a few problems:

- The libdnet library is called libdumbnet on Debian distributions. The libdnet library installs a DECnet system which we don’t need.
- The -dev libraries also had to be installed
- No C compiler was installed by default
- Also missing was; autoconf, automake, flex, bison, libedit, libedit-dev, zlib1g-dev, make, libc6-dev
- Error in file dhcpclient.c line 475 and 528

When running `honeyd`, we repeatedly got segmentation fault and `honeyd` shuts down. We also tried to install `Honeyd` by using `#apt-get install honeyd`. This also ended up with a segmentation fault.

A.2.2 Installing `honeyd` on Fedora Core 4

Downloaded honeyd-1.0a.rc2.tar.gz, libpcacp-0.9.4, libedit-0.3-1guru.suse100.x86.64.rpm. Used `#yum install <package>` to install the following packages:

- libdnet-1.10-2.fc4
- libevent-1.1a-2.fc4
- gcc
- flex
- bison
- libdnet-devel
- readline-devel
- zlib-devel
- compat-gcc-32

Then we ran `#export CC=gcc-32`, to make gcc32 the default compiler and not version 4. `honeyd` compiled OK, but still we got segmentation fault after a little while.

A.3 Solutions to the problems installing `honeyd`

It seems that the segmentation faults is regarding the configuration file that specifies the virtual hosts, and that there were missing some programs and libraries. When we tried to ping hosts which were not binded to specific IP addresses, the segmentation fault appeared. Addresses not binded are supposed to be answered with the default template in the configuration file, but this did not seem to work properly. Sometimes `honeyd` replied to unbinded addresses and sometimes not (mostly not). When we specified all addresses in the range, `honeyd` answered to ping, but it got segmentation fault after a while when real attackers attacked. The best we managed was with Fedora Core 3 installation with all development packets installed and updated. When running *honeyd-1.0a.rc2* on Fedora C3 without binding all addresses, we managed to make it work perfectly with attacks from the inside. When attackers from the outside send TCP packets to unbinded hosts, the segmentation fault again appears. We solved this by simply binding all addresses in the range we want to specific IP's. After doing this `honeyd` has run almost without segmentation faults. Now the segmentation fault appears seldom, thus making it possible to carry through an entire experiment without problems.

B WEB servers scan-rules (.50 SuSE)

These rules have been made by hand from the honeypot's weblog file, following the procedure presented in 6.6.3. Only the rules which created alerts in the control experient is reproduced. If the variabel $HTTP_SERVERS in snort.conf is used as intended, $HOME_NET should be replaced by $HTTP_SERVERS. This would speed up SNORT.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /login.htm";
flow:to_server,established; uricontent:"/login.htm"; nocase; priority:0;
sid:1000010; rev:1;)
```

Different from the original: [1564]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /robots.txt";
flow:to_server,established; uricontent:"/robots.txt"; nocase; priority:0;
sid:1000011; rev:1;)
```

Different from the original: [1852]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /CVS/Entries";
flow:to_server,established; uricontent:"/CVS/Entries"; nocase; priority:0;
sid:1000012; rev:1;)
```

Different from the original: [1551]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /Admin_files/";
flow:to_server,established; uricontent:"/Admin_files/"; nocase; priority:0;
sid:1000013; rev:1;)
```

Different from the original: [1212]

- uricontent:"/Admin_files";
- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /_vti_bin/";
flow:to_server,established; uricontent:"/_vti_bin/"; nocase; priority:0;
sid:1000014; rev:1;)
```

Different from the original: [1288]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /admin_/";
flow:to_server,established; uricontent:"/admin_/"; nocase; priority:0;
sid:1000015; rev:1;)
```

Different from the original: [1385]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /backup/";
flow:to_server,established; uricontent:"/backup/"; nocase; priority:0;
sid:1000016; rev:1;)
```

Different from the original: [1213]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /backups/";
flow:to_server,established; uricontent:"/backups/"; nocase; priority:0;
sid:1000017; rev:1;)
```

Different from the original: [1213]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /cgi-bin/";
flow:to_server,established; uricontent:"/cgi-bin/"; nocase; priority:0;
sid:1000018; rev:1;)
```

Different from the original: [1668]

- missing content:"/cgi-bin/ HTTP";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /doc/";
flow:to_server,established; uricontent:"/doc/"; nocase; priority:0;
sid:1000019; rev:1;)
```

Different from the original: [1560]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /iisadmin/";
flow:to_server,established; uricontent:"/iisadmin/"; nocase; priority:0;
sid:1000020; rev:1;)
```

Different from the original: [993]

- uricontent:"/iisadmin";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /iissamples/";
flow:to_server,established; uricontent:"/iissamples/"; nocase; priority:0;
sid:1000021; rev:1;)
```

Different from the original: [1402]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB intranet";
flow:to_server,established; uricontent:"/intranet/"; nocase; priority:0;
sid:1000022; rev:1;)
```

Different from the original: [1214]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /webalizer/";
flow:to_server,established; uricontent:"/webalizer/"; nocase; priority:0;
sid:1000023; rev:1;)
```

Different from the original: [1847]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB _mem_bin";
flow:to_server,established; uricontent:"/_mem_bin/"; nocase; priority:0;
sid:1000024; rev:1;)
```

Different from the original: [1286]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /dms0/";
flow:to_server,established; uricontent:"/dms0/"; nocase; priority:0;
sid:1000025; rev:1;)
```

Different from the original: [1872]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /oprocmgr-status/";
flow:to_server,established; uricontent:"/oprocmgr-status/"; nocase;
priority:0; sid:1000026; rev:1;)
```

Different from the original: [1874]

- uricontent:"/oprocmgr-status";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /server-info/";
flow:to_server,established; uricontent:"/server-info/"; nocase; priority:0;
sid:1000027; rev:1;)
```

Different from the original: [1520]

- uricontent:"/server-info";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /server-status/";
flow:to_server,established; uricontent:"/server-status/"; nocase; priority:0;
sid:1000028; rev:1;)
```

Different from the original: [1521]

- uricontent:"/server-info";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /srchadm/";
flow:to_server,established; uricontent:"/srchadm/"; nocase; priority:0;
sid:1000029; rev:1;)
```

Different from the original: [1040]

- uricontent:"/srchadm";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /test-cgi/";
flow:to_server,established; uricontent:"/test-cgi/"; nocase; priority:0;
sid:1000030; rev:1;)
```

Different from the original: [835]

- uricontent:"/test-cgi";

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /way-board/";
flow:to_server,established; uricontent:"/way-board/"; nocase; priority:0;
sid:1000031; rev:1;)
```

Different from the original: [896]

- uricontent:"/way-board";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /webcart/";
flow:to_server,established; uricontent:"/webcart/"; nocase; priority:0;
sid:1000032; rev:1;)
```

Different from the original: [1125]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /webcart-lite/";
flow:to_server,established; uricontent:"/webcart-lite/"; nocase; priority:0;
sid:1000033; rev:1;)
```

Different from the original: [1848]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /www-sql/";
flow:to_server,established; uricontent:"/www-sql/"; nocase; priority:0;
sid:1000034; rev:1;)
```

Different from the original: [887]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /~root/";
flow:to_server,established; uricontent:"/~root/"; nocase; priority:0;
sid:1000035; rev:1;)
```

Different from the original: [1145]

- uricontent:"/~root";
- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /DB4Web/";
flow:to_server,established; uricontent:"/DB4Web/"; nocase; priority:0;
sid:1000036; rev:1;)
```

Different from the original: [2060]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB PROPFIND /";
flow:to_server,established; content:"PROPFIND /"; nocase; priority:0;
sid:1000037; rev:1;)
```

Different from the original: [1079]

- content:"propfind";
- `pcre:"/<\x3a\s*propfind.*?xmlns\x3a\a*a=[\x21\x22]?DAV[\x21\x22]?/iR";`
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /mod_gzip_status/";
flow:to_server,established; uricontent:"/mod_gzip_status"; nocase; priority:0;
sid:1000038; rev:1;)
```

Different from the original: [2156]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB SEARCH /";
flow:to_server,established; content:"SEARCH /"; nocase; priority:0;
sid:1000039; rev:1;)
```

Different from the original [2091]:

- content:"SEARCH / HTTP/1.1|0D OA|Host|3A"
- content:"|0D 0A 0D 0A|"
- within:255;
- nocase not in original
- no reference
- no classtype:attempted-admin

```
 %samme regel som over
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB SEARCH /";
flow:to_server,established; content:"SEARCH /"; nocase; priority:0;
sid:1000039; rev:1;)
```

Different from the original [1070]:

- content:"SEARCH ";

- depth:8;
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /~root";
flow:to_server,established; uricontent:"/~root"; nocase; priority:0;
sid:1000051; rev:1;)
```

Different from the original: [1145]

- no reference
- no classtype:attempted-recon

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
 /<script>cross_site_scripting.nasl</script>"; flow:to_server,established;
uricontent:"/<script>cross_site_scripting.nasl</script>"; nocase; priority:0;
sid:1000040; rev:1;)
```

Different from the original: [1497]

- content:"<script>";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /NULL.ida";
flow:to_server,established; uricontent:"/NULL.ida"; nocase; priority:0;
sid:1000041; rev:1;)
```

Different from the original: [1242]

- uricontent:".ida";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /NULL.printer";
flow:to_server,established; uricontent:"/NULL.printer"; nocase; priority:0;
sid:1000042; rev:1;)
```

Different from the original: [971]

- uricontent:".printer";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /_vti_bin/
fpcount.exe"; flow:to_server,established; uricontent:"/_vti_bin/fpcount.exe"; nocase;
priority:0; sid:1000043; rev:1;)
```

Different from the original: [1013]

- uricontent:"/fpcount.exe";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
/_vti_bin/shtml.dll/_vti_rpc"; flow:to_server,established;
uricontent:"/_vti_bin/shtml.dll/_vti_rpc"; nocase; priority:0;
sid:1000044; rev:1;)
```

Different from the original: [940, 937]

- uricontent:"/_vti_bin/shtml.dll"; [940]
- uricontent:"/_vti_rpc"; [937]
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /iisadmpwd/aexp";
flow:to_server,established; uricontent:"iisadmpwd/aexp"; nocase; priority:0;
sid:1000045; rev:1;)
```

Need to use the LCS algorithm to find the longest common string from the six lines containing iisadmpwd.
Different from the original: [1018, 1487]

- uricontent:"/iisadmpwd/aexp2.htr"; [1487]
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB .htr";
flow:to_server,established; uricontent:".htr"; nocase; priority:0;
sid:1000046; rev:1;)
```

Need to use an algorithm which can see that the six lines containing iisadmpwd all ends with .htr.
Different from the original: [987, 1487]

- uricontent:"/iisadmpwd/aexp2.htr"; [1487]
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /search?NS-query-pat=
& /etc/passwd"; flow:to_server,established; content:"search?NS-query-pat="; nocase;
content:"/etc/passwd"'; nocase; priority:0; sid:1000047; rev:1;)
```

Different from the original: [1828]

- uricontent:"/search"; content:"NS-query-pat="; content:"../../";
- no reference
- no classtype:web-application-attack

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /_vti_inf.html";
flow:to_server,established; uricontent:"/_vti_inf.html"; nocase; priority:0;
sid:1000048; rev:1;)
```

Different from the original: [990]

- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB
/a.jsp/<SCRIPT>alert(document.domain)</SCRIPT>"; flow:to_server,established;
uricontent:"/a.jsp/<SCRIPT>alert(document.domain)</SCRIPT>"; nocase; priority:0;
sid:1000051; rev:1;)
```

Different from the original: [1497]

- content:"<script>";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /cgi-bin/
test-cgi.bat? |echo"; flow:to_server,established;
uricontent:"/cgi-bin/test-cgi.bat? |echo"; nocase; priority:0; sid:1000049; rev:1;)
```

Will not work because of the pipe symbol in uricontent string. This symbol is used when representing hexadecimal values, which echo is not.
Different from the original: [976]

- uricontent:".bat?";
- no reference
- no classtype:web-application-activity

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg:"NEW WEB /etc/passwd";
flow:to_server,established; content:"/etc/passwd"; nocase; priority:0;
sid:1000050; rev:1;)
```

Need to use the LCS Different from the original: [1122]

- no reference
- no classtype:attempted-recon

C Differences

Differences between Nessus traffic from the first attempt (control) and second attempt (new rules).

 # Diff web.log_webserverscan_50_control_org.txt web.log_webserverscan_50_newrules.txt > tmp
 # less tmp | grep GET > tmp2

```
< "GET /NonExistant759802178/ HTTP/1.1
> "GET /NonExistant397595117/ HTTP/1.1
< "GET /R4bgMqewQvLu.html HTTP/1.1
> "GET /rx8NoqOzN25_.html HTTP/1.1
< "GET /R4bgMqewQvLu.cgi HTTP/1.1
> "GET /rx8NoqOzN25_.cgi HTTP/1.1
< "GET /R4bgMqewQvLu.sh HTTP/1.1
> "GET /rx8NoqOzN25_.sh HTTP/1.1
< "GET /R4bgMqewQvLu.pl HTTP/1.1
> "GET /rx8NoqOzN25_.pl HTTP/1.1
< "GET /R4bgMqewQvLu.inc HTTP/1.1
> "GET /rx8NoqOzN25_.inc HTTP/1.1
< "GET /R4bgMqewQvLu.shtml HTTP/1.1
> "GET /rx8NoqOzN25_.shtml HTTP/1.1
< "GET /R4bgMqewQvLu.asp HTTP/1.1
> "GET /rx8NoqOzN25_.asp HTTP/1.1
< "GET /R4bgMqewQvLu.php HTTP/1.1
> "GET /rx8NoqOzN25_.php HTTP/1.1
< "GET /R4bgMqewQvLu.php3 HTTP/1.1
> "GET /rx8NoqOzN25_.php3 HTTP/1.1
< "GET /R4bgMqewQvLu.cfm HTTP/1.1
> "GET /rx8NoqOzN25_.cfm HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.html HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.html HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.cgi HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.cgi HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.sh HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.sh HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.pl HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.pl HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.inc HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.inc HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.shtml HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.shtml HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.php HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.php HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.php3 HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.php3 HTTP/1.1
< "GET /cgi-bin/R4bgMqewQvLu.cfm HTTP/1.1
> "GET /cgi-bin/rx8NoqOzN25_.cfm HTTP/1.1
< "GET /~IOWiMTNw HTTP/1.1
> "GET /~itSNqTkz HTTP/1.1
< "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cautoexec.bat1842598681 HTTP/1.1
> "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cautoexec.bat1757841550 HTTP/1.1
< "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cwinnt%5cwin.ini958608582 HTTP/1.1
> "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cwinnt%5cwin.ini1599968663 HTTP/1.1
< "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cboot.ini1944742498 HTTP/1.1
> "GET /error/%5c%2e%2e%5c%2e%2e%5c%2e%2e%5c%2e%2e%5cboot.ini745367903 HTTP/1.1
```

This shows that the only differences are that numbers and some of the strings consist of randomized numbers or letters. In the original file there are approximately 830 lines. As the `diff` command shows there are only 24 lines in the new experiment file which

differ from the original. This is 2,89% difference. None of these differences have any affect on the experiment result.