Helsinki Metropolia University of Applied Sciences
Degree Programme in Information Technology

Evgeniy Arbatov

Development of Hybrid Honeynet for Malware Analysis

Bachelor’s Thesis. 19 April 2010
Supervisor: Erik Pätynen, Senior Lecturer
Language Advisor: Taru Sotavalta, Senior Lecturer

Helsinki Metropolia University of Applied Sciences **Abstract**

Author	Evgeniy Arbatov
Title	Development of Hybrid Honeynet for Malware Analysis
Number of Pages Date	58 19 April 2010
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Supervisor	Erik Pätynen, Senior Lecturer

The goal this project was to develop a system that would be capable of supplying application layer context of one host for the network layer traces collected by a second independent host. The aim of the resulted system, based on the principle of decoys, would be to analyze potentially malicious traffic without a threat to production services.

The practical part of this work involved building a scalable setup to verify the feasibility of the chosen methods. The study concentrated primarily on the Linux Slapper worm and used a sample of the worm's exploit code to develop a mechanism for attracting a worm and creating a signature suitable for the worm identification by an IDS system. The key outcomes of the project included establishing the honeynet environment for inspecting malicious traffic flows and examining the worm functionality. The work also reflected on the body of knowledge presently available by studying the different solutions in the field of malware analysis.

The developed system has no production value, but provides an outlook into functioning and interfaces between various technologies that usually operate independently. Further study could include examination of low-level components and assumptions made by software developers and malware authors. Additional work on formalizing the properties and modelling worm interaction could also assist in improving the understanding of malicious code.

Keywords	honeypot, honeynet, IDS, worm, virtualization, HTTPS, heap overflow, shellcode, sebek

Contents

Abstract 2

Abbreviations 5

1 Introduction 6

2 Previous Studies on Honeypots and IDS Systems 7

- 2.1 Properties of Computer Worms 7
- 2.2 Honeypots and Related Technologies 8
  - 2.2.1 Understanding Honeypots and Honeynets 8
  - 2.2.2 Present High-interaction Honeypot Models 10
  - 2.2.3 Existing Honeynet Architectures 11
  - 2.2.4 Summary of Virtualization Principles 13
- 2.3 Intrusion Detection Systems 14
  - 2.3.1 IDS Principles 14
  - 2.3.2 HIDS Context to Aid Network Layer Visibility 16
  - 2.3.3 Innovative Approaches to Intrusion Detection 17
- 2.4 Automated Code Analysis 18
  - 2.4.1 Analysis of Exploit Payload 18
  - 2.4.2 Automated Worm Response 19
- 2.5 Attacks on IDS and Honeypot Systems 21
  - 2.5.1 Defeating an IDS 21
  - 2.5.2 Low-interaction Honeypot Reconnaissance 21
  - 2.5.3 High-interaction Honeypot Discovery 22
  - 2.5.4 Addressing Delectability Issues 23

3 Honeynet Development and Worm Analysis 25

- 3.1 Project Context 25
- 3.2 Honeynet Design 25
- 3.3 Virtualization Setup Options 26
  - 3.3.1 VMware ESX Configuration 27
  - 3.3.2 Setting Xen Virtualization 27
- 3.4 Configuring Individual Honeynet Hosts 28
  - 3.4.1 Low- and High-interaction HTTP Servers 28
  - 3.4.2 Honeynet Gateway Setup 29
  - 3.4.3 Setting Covert Monitoring 33
- 3.5 Verifying Honeynet Settings with Nmap 34

3.6 Study of Apache Worm 35
3.6.1 Debugging Apache HTTP Server and SSL Plugin 35
3.6.2 Understanding Shellcode Used 37
3.6.3 Preparing Exploit Code 38
3.6.4 Web-server Exploitation Technique 40
3.6.5 Heap Overflow: Low Level Perspective 41
3.7 IDS Signature Generation 43
3.7.1 Scripting of Signature Creation 44
3.7.2 Existing Signature in Snort IDS Ruleset 45

4 Results 46

5 Discussion 48

6 Conclusion 51

References 52

Appendices

Appendix 1: Script for Extracting Hex Dumps from Pcap Files 56
Appendix 2: Script for Measuring Snort IDS Signature Execution Time 58

Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BOOTP	Bootstrap Protocol
CPU	Central Processing Unit
DLL	Dynamic-link Library
DomU	Unpriveleged Domain in Xen
DoS	Denial of Service
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GOT	Global Offset Table
GRE	Generic Routing Encapsulation
GUI	Graphical User Interface
HIDS	Host-based Intrusion Detection System
HIH	High Interaction Honeypot
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HVM	Hardware-assisted Virtualization
IDS	Intrusion Detection System
IDT	Interrupt Descriptor Table
I/O	Input/Output
IP	Internet Protocol
IPv6	Internet Protocol version 6
LIH	Low-interaction Honeypot
MAC	Media Access Control
NAT	Network Address Translation
NIC	Network Interface Controller
NIDS	Network-based Intrusion Detection System
OS	Operating System
OUI	Organizationally Unique Identifier
P2P	Peer-to-peer
PCAP	Packet Capture
SELinux	Security-Enhanced Linux
SSL/TLS	Secure Sockets Layer / Transport Layer Security
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
UML	User-mode Linux
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 Introduction

The goal of this project is to study an automated system for examining malicious activity and creating network Intrusion Detection System (IDS) signatures. The purpose is to obtain a generic solution with flexibility of an anomaly-based IDS and effectiveness of a signature-based IDS. Automation is also an important goal, because attempting to produce more autonomous IDS systems results in gaining better understanding and facilitating the manual process of signature creation.

The focus of this study is on autonomously spreading malware or worms that typically present a challenge for effective identification and containment due to high propagation rates. The major project objective is to compare the efficiency of signatures produced by the constructed honeynet system, passively intercepting and processing malicious flows, and those made by an IDS vendor. Producing sound security tools, requires knowledge of how a potential attacker operates. Therefore in the course of my project I will also consider the implementation of the relevant Apache exploit.

The scope of this project is limited to developing a system that is capable of automatically performing the task of attracting and capturing malware. No heuristics or machine learning is being implemented. The system is limited to working with Hypertext Transfer Protocol (HTTP) and HTTP Secure (HTTPS) protocols and a single worm only. While limited in scope, my project contains a mix of independent technologies, which, if made aware of one another, will be capable of greater precision and lower cost at identifying malicious patterns.

2 Previous Studies on Honeypots and IDS systems

2.1 Properties of Computer Worms

According to Mell, Kent and Nusbaum (2005) malware or malicious software is "a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability" [1]. Examples of malware include viruses, worms, Trojan horses, backdoors and rootkits. Worms, and in particular network service worms, are of most interest in this work. Worms being self-contained, self-replicating and self-propagating malware pose a significant challenge, because human reaction is too slow to prevent wide-spread infections.

For spreading over the Internet, worms use a variety of scanning techniques to locate vulnerable hosts. The speed of a worm spreading depends on scanning methods used by a worm. The well-known worm scanning methods include:

- random scanning: searching the entire address space
- routable scan: targeting only reachable subnets
- hit-list scanning: using a list of vulnerable hosts initially and random scanning afterwards
- divide and conquer scan: avoiding re-scanning by dividing address ranges between worm instances
- local preference scanning: hosts in the same locality are likely to have similar vulnerabilities

Random scanning is simple, but highly inefficient, especially with Internet Protocol version 6 (IPv6) deployments. Routable and hit-list scanning require a worm to carry pre-compiled address ranges, increasing the size and limiting impact of the worm. The divide and conquer scan is the first step at making worms coordinate their scanning tactics. Local preference scanning is an example of the method using non-uniform distribution of vulnerable hosts in a particular domain name, autonomous system or IP

range. The more robust propagation can be achieved by worms with properties of collective intelligence. An idea analysed by Yi and Xiangning (2006) shows how newly created worms can communicate with a creator worm to modify their propagation behaviour, analogous to ant colonies [2], which can be viewed as a type of divide and conquer scan.

The work of Chuanyi and Chen (2005) on enhancing worm efficiency demonstrates benefits of importance scanning, or reducing the number of worm probes based on distribution of vulnerable hosts, for producing a worm that is able to modify its attack patterns, based on distribution of vulnerable host population. The idea implies a need for having a central location for collecting and computing probabilities, while at the same time keeping a low level of information exchange. To get the host distribution the worm first has to use random scanning, which can be slow with IPv6 addresses, but is made more effective by applying importance scanning algorithms. Defences against such worms consist of deploying applications uniformly, difficult due to inherent non-uniform structure of the Internet, and information exchange between different network domains, to stop the worm during learning stage. [3]

2.2 Honeypots and Related Technologies

2.2.1 Understanding Honeypots and Honeynets

A honeypot is a tool that is set up for the purpose of being attacked and compromised. Since there is no production function for a honeypot, any interactions with the honeypot usually indicate a malicious activity. Therefore, previously unknown attacks can be discovered.

Honeypots exist in two distinct categories: high-interaction and low-interaction. Low-interaction honeypots (LIH) use scripts to emulate Transmission Control Protocol (TCP) based services. If services are emulated, there is no possibility of exploiting a particular service, thus risks are minimized. LIH honeypots use fewer resources and are

able to emulate complete network topologies. High-interaction honeypots (HIH) on the other hand run vulnerable services without emulation. If a HIH honeypot is compromised, an attacker will gain full access to the resources of a vulnerable machine.

Honeynets are research honeypots that are composed of HIH honeypots. Honeynets evolved from being run on separate physical machines and presently make use of virtualization. Two major requirements of the honeynet are data control and data capture. Data control is offered by a Layer 2 device, honeywall, that monitors outgoing and incoming traffic. The control of outgoing traffic can, for example, be performed by using Snort-inline to overwrite harmful contents of an outgoing packet. However, this relies on the fact that an IDS is already aware of the particular attack pattern. This is not always true, since one goal of honeynets is to discover previously unknown attacks. Data capture is performed at several layers: on the honeywall and on the HIH honeypot host. The latter is needed, when an attacker is using encrypted tunneling for communicating with the honeypot.

The definition of hybrid virtual honeypots has two meanings:

- honeynet setup with virtualized HIH honeypots and physical host running honeywall
- honeynet combining virtualized honeywall, LIH and HIH honeypots.

During my work I made use of the second definition that includes different honeypot types with all the components of the honeynet being virtualized. One of the main arguments for running a honeywall on a separate physical host is additional security gained for the honeywall, which is not affected by malicious activity on HIH honeypots. However, exploits targeting virtualization layer are rare, and therefore it is safe to run a completely virtualized honeynet.

The honeypots are not necessarily passive monitoring devices. The development of client-side honeypots resolves the issue of resource idleness and establishes a more proactive approach to end-user security. Typical client honeypots exist in a form of an emulated web browser that makes queries to various potentially malicious servers, attempts to circumvent obfuscation techniques used by malware and execute the resulting code to analyze the level of a threat. Examples of client honeypots are HoneyC and Microsoft's closed-source Honeymonkey [4].

The challenges of server-side honeypots are distinct from those of client-side honeypots. For example, server-side honeypots may create an illusion of a suspiciously homogeneous environment. Another challenge is the stationary location of honeypot systems. Thus a production server can still be compromised, without a honeypot being notified. LaBrea honeypots take a different approach by occupying unused address space and tarpitting or slowing down the spread of malware, a technique that can be useful in defense against worms and spammers.

An important development in honeypotting technology is the ability of distributed honeypots to communicate with each other and correlate information obtained from different sensors, similarly to the operation of distributed IDS systems described in section 2.3.3.

2.2.2 Present High-interaction Honeypots Models

Several approaches have been taken to study attacker's behaviour with HIH honeypots. Yan (2005) used User-Mode Linux (UML) to provide virtualized environment for controlled interaction with malware. The major advantage of using the UML is access to monitoring of processes executed in a guest OS from a host OS and logging keystrokes, which is extended with a file system imaging tool, capable of taking snapshots of a guest OS for further examination. Security-Enhanced Linux (SELinux) is also used to control processes executed inside a guest OS, relying on visibility of the guest OS properties from a host OS. Yan (2005) also argues that visibility of UML-

based virtualization in honeypots does not present a major threat, because of virtualization becoming wide-spread. [5]

Briffaut, Lalande and Toinard (2009) do not make use of virtualization for capturing malicious activities with HIH honeypots, instead a cluster of different physical hosts is used. Although, different platforms are employing different logging standards, the study highlights the benefits of correlating network and host activity in order to identify effects of network traffic on recipient systems. Hosts based on SELinux proved to be the most attack-resistant and did not need re-installation despite numerous exploits being attempted, mainly because root-user privileges are limited by SELinux policies. The whole system, however, needs to be operated manually, except for the automated re-installation of infected hosts, using Bootstrap Protocol (BOOTP) and Trivial File Transfer Protocol (TFTP). [6]

The intrusion detection framework by Artail, Safa and Sraj (2006) describes an adaptive structure consisting of LIH and HIH honeypots. The work addresses the primary limitation of many honeypot-based solutions, i.e. not following the distribution of OS versions and services in the rest of the network. The proposed solution is highly scalable as it makes use of the unoccupied Internet Protocol (IP) addresses for LIH honeypots that redirect traffic to HIH honeypots. The work of Artail, Safa and Sraj (2006) stresses that the availability of a large number of IP addresses is a key to effective honeypot-based systems. Having created a distributed honeypot model, the framework does not provide means for signature generation and is not capable of preventing ongoing intrusions. Instead a site administrator is notified about malicious activity. [7]

2.2.3 Existing Honeynet Architectures

Performance and a need to obtain high fidelity attack information have driven the development of large-scale hybrid honeynet architectures or honeyfarms. The idea of traffic redirectors from an attacked host to a central analysis and correlation centre is presented in Collapsar project by Xuxian, Dongyan and Yi-Min (2006). By means of

centralizing the traffic analysis, the processing is offloaded from end-user systems. However, challenges arise with identification of the traffic that needs redirection and methods for traffic redirection. In Collapsar the proposed solutions consist of application level redirectors or modifications to routing topology for suspicious traffic. [8]

A further step in maximizing the number and efficiency of HIH honeypots is done with the development of Potemkin honeyfarm by Vrable, Ma and Chen (2005).This architecture is based on the principle of spawning a virtual machine only when a particular type of activity is detected. Potemkin developers propose creating a reference memory snapshot of a HIH honeypot to reduce startup time. A honeyfarm gateway is then responsible for management of resources and containment. [9]

Unfortunately none of the scalable honeyfarms is available as an open-source implementation, although some of the proposed solutions provide useful directions for future developers. The Potemkin and Collapsar honeyfarms make use of Generic Routing Encapsulation (GRE) tunnels to route potentially malicious traffic. Xen and UML Linux are the primary virtualization platforms, due to modifications of a Virtual Machine Monitor (VMM) needed to process requests from the honeyfarm gateway, as well as extension of monitoring facilities.

While the proposed large scale solutions are potentially able to provide scalability, it is unclear which traffic should be redirected to centralized HIH honeypots and how to maintain images of virtual machines in case of long-term attacks. Another major challenge is addressing the timing delay of sending potentially malicious traffic to a honeyfarm and responding back to an attacker. Hiding the presence of a traffic redirector is not a trivial task either.

2.2.4 Summary of Virtualization Principles

At the heart of deploying a scalable honeypot solution lie the benefits provided by virtualization. Virtualization is the ability of the physical server hardware to be shared by several virtual machines. Among different virtulization methods only full-virtualization and paravirtualization are feasible in the implementation of HIH honeypots. The major benefits provided by these virtualization methods are resource isolation and a possibility of restoring a virtual machine (VM) to a past reference image.

An example of a paravirtualized VMM is Xen [10]. Xen is able to effectively and securely share available resources, while giving a guest VM an ability to see real hardware resources. One of the added benefits is the increased speed of execution by making a guest VM aware of virtualization layer, which involves modification of the guest OS kernel, although no changes are needed for applications running in the modified guest OS.

Full virtulazation allows running a guest OS unmodified and provides a set of generic hardware devices to be accessed by the guest VM. In particular, using VMware ESX provides full-virtualization support, also allowing to run a VMM without the underlying host OS. Hardware independence of full virtualization solutions and portability of guest VMs leads to wide spread deployment of such systems, particularly in connection with closed-source applications and end-user systems.

Although virtualization is an important part of running enterprise-level services, x86 architecture was not designed to support virtualization. As a result of concurrently running several virtual machines, there exist sensitive instructions that must not be executed directly by a processor. Additionally, running guest VMs on a lower privilege level than a VMM results in several privileged instructions needing emulation by the VMM. However, the x86 instructions set contains several sensitive unprivileged instructions, not emulated by the VMM. Thus a complete isolation of guest VMs is not achieved. [11] Hardware assisted virtualization for x86-based processors have been

developed with the introduction of Intel-VT and AMD-V. With hardware-assisted virtualization a processor is using hardware extensions to emulate sensitive instructions.

The VMM creates a new attack surface, posing threats such as resource mapping and insufficient fault isolation. Interestingly, virtualization technology by itself can also be used by attackers. The BluePill project has developed a nested virtualization solution for placing an attacker's code into a VM, that is isolated from an attacked computer. Although detection of an unauthorized virtualization layer can be performed, it is not currently possible to detect the presence of BluePill. [12]

2.3 Intrusion Detection Systems

2.3.1 IDS Principles

To examine traffic passing through firewall rules and protect servers and end-user systems from ever-present software bugs, intrusion detection systems (IDS) need to be deployed in an enterprise network. IDS is defined as a mechanism to "detect violations of system security policy" [13]. IDS systems are classified into a host- and network-based IDS. The Host IDS (HIDS) is working by means of auditing a single host, while the network IDS (NIDS) is monitoring network traffic flowing between hosts. The most important features of an IDS include:

- running continuously without being supervised
- fault tolerance or ability to recover from crashes
- monitoring IDS consistency and self-checking against compromises
- adapting to changes in user and network behaviour
- graceful degradation or ability to continue working if some of the components fail
- configurable to reflect needed security policies. [14]

IDS systems are typically divided into anomaly-based and misuse-based systems. The misuse IDS relies on existence of patterns or signatures matching malicious behaviour, similar to signature based anti-virus products. Misuse IDS is not able to identify attacks not matching a set of defined rules.

The anomaly IDS detects deviations from normal activity expressed in terms of network flows, user profile or application use. Such an IDS can detect unknown attacks, given that the normal activity estimate is accurate and the normal versus abnormal threshold is able to adapt to changing user needs. Anomaly detection can be separated into static and dynamic anomaly detection. A static anomaly detector relies on some constant property of a system, for example, OS code, whereas a dynamic anomaly detector uses event records and network traffic data. [13] Anomaly IDS systems are frequently based on various machine learning techniques, such as pattern classification, single classifiers and generic algorithms.

The output of an IDS, correctly identifying an attack is called a true positive, while treating malicious activity as normal is called a false negative. A false positive occurs when normal activity is seen as an attack, whereas a true negative correctly classifies non-malicious activity. Detection rate and false alarm rate are some of the metrics used to evaluate IDS performance:

$$Detection\ rate = \frac{True\ postive}{True\ positive + False\ negative}$$

$$False\ alarm\ rate = \frac{False\ positive}{True\ negative + False\ positive}$$

The Snort IDS is a misuse-based network IDS. The Snort IDS was chosen for this project, because of being open-source, as well as extensible, and having a developed community and documentation. Figure 1 illustrates the major components of the Snort IDS.

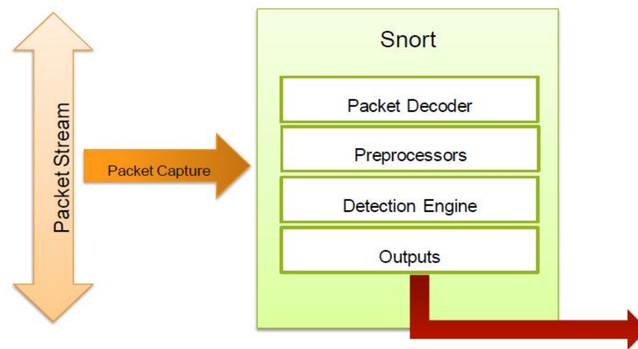

Figure 1. Snort IDS architecture. Reprinted from Writing Effective Rules, Part 1 (2008) [15]

As can be seen from Figure 1, the first step of processing incoming packets is the packet decoder. Packet decoders perform input data parsing to identify separate packets and their fields: headers and payload. Preprocessors prepare packets for detection engine by reassembling and normalizing packets, to represent data in a standardized manner. [15] The detection engine and output formatting take care of Snort IDS rule matching and result formatting.

2.3.2 HIDS Context to Aid Network Layer Visibility

A technique proposed by Rabek, Khazan and Lewandowski (2003) of using the static analysis of executables to locate Win32 API calls and monitor the calls made to the API during program runtime. Such an approach is effective against malware that is trying to hide itself by means of injection, polymorphism or obfuscation, but cannot deal with malware not using such techniques or cause false positives on software protection and the legitimate use of dynamic API binding. This framework works in two stages: preprocessing, to locate APIs and record return addresses, and validation, to verify an instruction for an API and the return address. To successfully detect malicious activity, the framework needs for malware to interact with Win32 API and not only exploit the target software. [16]

Combining views seen by HIDS and NIDS has been studied by Dreger, Kreibich and Paxson (2005) for adding host context information to an IDS looking at traffic flows, which would allow for a NIDS to analyze the exact protocol state of an end-host and observe events over encrypted channels. The Bro IDS has been used for receiving application specific events notifications from end-hosts and comparing the results seen by an end-host with the ones 'seen on the wire'. The solution proved to have a good response time and can even allow removing application level decoding from an NIDS, but a challenge is in recording manipulations by an end-host upon receiving a client request, such as URL-canonicalization. [17]

2.3.3 Innovative Approaches to Intrusion Detection

Traditionally, IDS systems are deployed within the limits of a certain organization. Hence there are presently no commercial solutions that would enable effective sharing and correlation of alerts obtained from IDS systems located in different administrative domains. This is one of the challenges addressed by the research by Locasto, Parekh and Keromytis (2005) aimed at creating a distributed and collaborative IDS. In their work Locasto, Parekh and Keromytis (2005) identify the reliability of information sources for IDS alerts and integrity and confidentiality of exchanged data as challenges to cross-organisational sharing of IDS alerts. Use of Bloom filters, one way hash-functions, allowing only verification and insertion, is proposed to avoid leaks of confidential information via alert exchanges. A distributed correlation function is being used for correlating alerts from several sources to avoid having a single point of failure with a centralized storage service. An important achievement of such a solution is the ability to detect low-volume scanning, as well as wide reaching outbreaks. [18]

A different approach to cope with IDS visibility problems is discussed by Garfinkel and Roseblum (2003), who demonstrate an implementation of VMM-based HIDS, allowing better access control and isolation of an IDS from the monitored host. An assumption is

made that VMM is a relatively straightforward component and thus is not vulnerable to compromises. To be effective, an IDS resides in a separate VM and monitors only potentially hazardous calls to avoid performance penalties. Hence an attacker capable of modifying, for example the interrupt dispatch table (IDT) will mislead an IDS about the true state of the monitored system. By controlling access to sensitive memory areas an IDT table can be secured, although several dynamic kernel structures in the memory cannot be protected by a developed implementation. [19]

2.4 Automated Code Analysis

2.4.1 Analysis of Exploit Payload

A major drawback of automated analysis systems is that such systems study only a single execution path and are not able to observe all conditions which can be seen by means of the static analysis. This limitation had been addressed by the study of Moser, Kruegel and Kirda (2007) on developing a system capable of detecting conditional execution paths. The proposal includes the idea to dynamically track input values requested by malware samples, to identify a code branch, to take a snapshot of a state before the branch and to run the sample by selecting different condition inputs. [20]

CWSandox is one of the tools for dynamic malware analysis of malware samples. CWSandox relies on Application Programming Interface (API) hooking, in-memory overwriting of the call to Windows API with an address of a hook function, which will perform a Dynamic-link Library (DLL) injection by directing the program flow to a customized DLL code. [21] However, if malware is able to verify its own integrity or the integrity of the supplied DLLs, or make use of its own DLLs, CWSandox will not be able to detect such malware.

2.4.2 Automated Worm Response

The study by Moore, Shannon and Voelker (2003) identifies prevention, treatment and containment as properties that affect the number of hosts infected by a worm. By following secure software development practices it is possible to minimize the vulnerabilities present and slow down worm propagation. Treatment relates to the amount of time needed for an analyst to identify a worm and produce a reliable signature. Containment involves the actual measures taken to prevent worm spreading. Estimated reaction time for defeating worms with content filtering is one to two hours [22]. Major conclusions of the work by Moore, Shannon and Voelker (2003) are the need for automated techniques of identification and response to worm spread and content filtering up to the application layer to effectively filter out worm traffic.

Several properties for signatures generated to discover malicious activity have been identified [23;24]. Signature quality depends on the ability of the signature to accurately distinguish malicious and non-malicious activities. The signature number and length of the needed pattern matching affects the performance and throughput of an IDS system. Another property of IDS signatures is effectiveness against polymorphism, presently based on the non-idealities of polymorphic generators, such as having identical worm decryptor routines. The timeliness of obtaining signatures and application independence of the created signatures are also important properties of effective threat mitigation.

Previous attempts to create a system to respond automatically to worm threats include the development of Autograph [23], a system for automated signature generation for TCP-based services, relying on identification of the most frequently occurring patterns in a TCP flow sample to formulate worm signatures. Such an assumption is valid, because while a worm is spreading, repeated byte patterns can be identified in an affected network segment. Autograph limitations are a slow process of the TCP flow

reassembly and inability to detect worms, using hit-list scanning and thus having a minimal number of identical packets in network flows.

Earlybird does not rely on honeypots, but is detecting worms by identifying characteristics inherent to worms, such as invariable strings across all the worm instances and uniform distribution of network traffic from infected hosts [24]. Earlybird is designed to generate signatures for NIDS, such as Snort and Bro, and has a throughput of 200 Mbit/s, and is able to detect a wider range of worms by not using traffic pre-classification, employed by Autograph. To reduce the number of false positives, traffic matching signatures generated by Earlybird can be rate-limited and monitoring continued to identify if a suspicious traffic reaches a threshold to be identified as malicious. [24]

Honeycomb is a tool for automatic generation of signatures for NIDS by means of pattern matching and protocol analysis, based on Honeyd [25]. Good IDS signatures should be specific, while capturing variations of the same attack. By using the longest common substring algorithm Honeycomb creates signatures, attempting to find matches between honeypot and IDS traffic. [25]

Honeybow is one of the closely related projects with an aim of automatic collection of autonomous malware [26]. Honeybow relies on monitoring network and file system activity for detecting malware. Honeybow targets only Windows systems and makes use of a tool for extracting PE executables, PE hunter, with Snort IDS from the network stream. The main disadvantage of Honeybow is an easily detectable program for real time monitoring of changes to a file system, a drawback which is reduced by analysis tools existing outside of a honeypot. [26]

A step towards automated worm containment, as opposed to detection and prevention, has been made by the Virus Throttle project, which exploits differences between normal and malicious network activity of a usual end-user system [27]. The idea is based on

keeping track of connections made by an end-user. Thus if a host is infected, it will attempt to make high-rate connection attempts to new, previously unrequested destinations. By means of analyzing code behaviour, instead of performing matching with signatures, Virus Throttle can be effective in detecting zero-day attacks. Finally, Virus Throttle reduces the risk of false positives, since a legitimate connection can only be delayed, but not blocked. [27]

2.5 Attacks on IDS and Honeypot Systems

2.5.1 Defeating an IDS

An attacker has several ways to detect the presence of an IDS in a target network. For example, reverse DNS queries on an attacker's IP address or scans launched by an attack target mean that the IDS is collecting information for intrusion alert information. An IDS/IPS system may also change firewall rules in response to a protected target being scanned or probed. Some passive, not inline IDS systems, will forge RST packets, which can be seen by looking at packet headers. [28]

Once an IDS is detected, a careful attacker will try to avoid being noticed. An easy access to the rules published, for instance, by Sourcefire for Snort, makes it trivial for an attacker to learn which features the Snort IDS is looking for. Techniques such as slowing down the scanning, avoiding sequential scans of network ranges and packet fragmentation are widely known IDS-avoidance tactics. Spoofing an attack source and DNS proxying insure that an attacker's identity will remain unknown to a compromised party. [28]

2.5.2 LIH Honeypot Reconnaissance

To detect an LIH honeypot, an adversary does not need to compromise the honeypot. Observing inconsistencies in services being executed, such as Windows Server application on a Linux host, noticing correlation between load on one LIH honeypot and

response time of other seemingly unrelated machines are among the most trivial honeypot detection techniques. Additionally, by crafting invalid packets or looking at responses to fragmented packets it is also possible to detect emulated honeypot environments. [29] By sending malformed SYN packets to older versions of Honeyd and getting an RST response, it is possible to distinguish between a Honeyd host and a real one.

2.5.3 High-interaction Honeypot Discovery

Honeypot technology benefits that make honeypots a fruitful resource for network defenders can also be used by attackers to detect monitored environments. The detection tools can be as generic as measuring timing of malware execution and debugger detection to sophisticated honeypot-specific techniques. By creating hitlists with honeypot addresses, noticing configuration differences between production systems and honeypots, and observing that honeypot technologies are mostly used for servers, an attacker is able to pinpoint and avoid honeypots.

Virtualization is invaluable for implementing honeypots that are not used in production, but can be easily spotted. For example, VMware can be detected by looking at the VMware-specific names of network and audio cards, Windows registry or SCSI/IDE devices. Additionally, by obtaining the Organizationally Unique Identifier (OUI) of a Media Access Control (MAC) address it is trivial to trace a VMware network interface card (NIC). Moreover, VMware has an I/O-backdoor allowing to manipulate the virtual machine clipboard and get information on the VMware version, the contents of a clipboard or CPU frequency. [30] To make use of these VMware features, an attacker will need to load a known magic number into the EAX register and pass call parameters in EBX and ECX [31].

The Sebek covert monitoring, based on a principle of Linux kernel module rootkits overwriting system calls for reading or opening files and creating network sockets, can

be detected relatively easily. There are several techniques to identify the presence of Sebek. Since Sebek relies on the method of covertly sending User Datagram Protocol (UDP) packets over the network, by causing multiple calls to the read() syscall, it is possible to create network congestion and a delay detectable with a ping command.

Also by comparing the number of transmitted packets seen with different methods on Linux systems and examining the memory for Sebek structures, the Sebek presence can be identified. Since Sebek needs to modify the read() syscall, by checking the location of write() and read() syscalls in the memory one is able to see an abnormally large distance between the two, indicating Sebek module being loaded. The tool, Kebes, developed by Holz, Dornseif and Klei (2004) allows an attacker to establish an encrypted tunnel and avoid the use of monitored Sebek read() syscall completely. [32]

The detection of a honeywall, running Snort-inline in front of HIH honeypots, is done by observing restrictions on traffic originated from the honeypots and packet rewriting performed to sanitize malicious content originated from a honeynet.

2.5.4 Addressing Delectability Issues

The visibility of honeypot techniques is one of the major challenges addressed in several studies. Research done by Song and Takakura (2008) proposes a method for intelligent port opening to protect from the discovery of honeypots based on low-interaction Nepenthes, by default listening on all ports for observing attacker's activities [33]. A study conducted by Khattab, Sangpachatanaruk and Mosse (2004) makes an attempt to hide honeypots in a pool of legitimate servers, proposing a mechanism of notifying legitimate clients to switch from one server to another. By means of such a method, attack traffic can be either reactively directed to honeypots or proactive measures can be taken to prevent unknown attacks. This solution, however, poses an overhead, acknowledged by Khattab, Sangpachatanaruk and Mosse (2004), of having

clients to reestablish active TCP connections and reduction in available server processing resources, caused by introduction of honeypots in a server spool. [34]

3 Honeynet Development and Worm Analysis

3.1 Project Context

The system for automated capture and analysis of autonomous malware was set up in the networking laboratory of the Helsinki Metropolia University of Applied Sciences, Espoo, Finland. This work is a result of part-time activity over a period of eight months, while learning the essential skills of system administration and getting hold of the basic concepts in malware research. This project relies on freely available material found on the Internet, research done by the different chapters of the Honeynet project and relevant mailing lists. My contribution with the present project is collecting all the needed information and reflecting on the available solutions to build my honeynet system.

3.2 Honeynet Design

Figure 2 illustrates the complete honeynet setup as seen from a high level. The honeynet was run on the VMware ESX server, with the virtual switch being a part of the VMware ESX server.

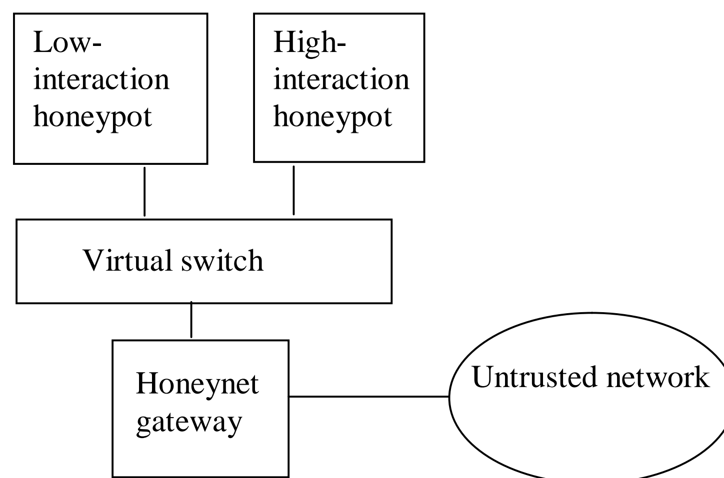

Figure 2. Overview of honeynet setup

The honeynet consists of the following components:

- honeynet gateway with signature generation script
- LIH honeypot
- HIH honeypot

The honeynet gateway was responsible for making decisions on attack traffic redirection between LIH and HIH honeypots. The signature generation script processed traffic dumps of captured honeynet sessions, analyzed information from HIH honeypots and extracted relevant packets from the traffic dump to be placed in a signature.

The LIH honeypot was the front-end of the honeynet setup. Every connection was established to the LIH honeypot at first, thus allowing to select only unique packets for sending to HIH honeypots. The LIH honeypot was running an emulated copy of a web server on the HIH honeypot. The HIH honeypot accepted redirected connections from the honeynet gateway and allowed an attacker to interact with a non-emulated HTTP server. Thus a more definite identification of malicious traffic was performed.

The honeywall was not present in this setup, because although it did provide a convenient web interface, there was no ready-made command line utilities to extract the tcpdumps from Honeywall databases. Instead a less scalable solution to capture traffic directly to the packet capture (PCAP) files on the honeynet gateway was used.

3.3 Virtualization Setup Options

In the course of my project I used VMware ESX virtualization, because most of the software supplied by the Honeynet project was not aware of virtualization, thus requiring running of unmodified guest VMs. Although, Xen with Hardware-assisted virtualization (HVM) support could be an alternative to VMware ESX, the Intel Xeon processor I used did not support Intel-VT. Hence running unmodified guest VMs with

Xen is not possible. UML Linux and Qemu were also not suitable either, because they required modifications to the guest OS.

3.3.1 VMware ESX Configuration

Guest VMs for VMware ESX are available from VMware Marketplace, with a choice of Debian 4.0 and Debian 5.0 images. By using those images and VMware Converter the guest images can be transferred to the VMware ESX server. VMware ESX networking is done with two switches, created via VMware Perl API scripts. The default switch policies are appropriate. Figure 3 shows how the interconnections between the switches and guest VM are done.

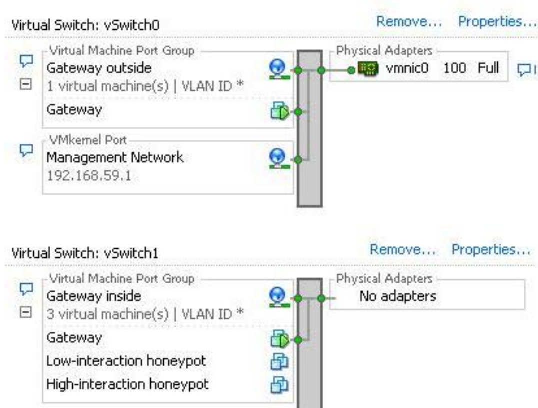

Figure 3. Honeynet networking in VMware ESX server

As Figure 3 demonstrates, the connections from LIH and HIH honeypots are restricted and subject to policies applied on the honeynet gateway.

3.3.2 Setting Xen Virtualization

In contrast to the VMware ESX server, there are no precompiled images available for Xen. However, using tools, such as xen-create-image, new Debian VMs can be installed with minimal effort. For example, by issuing the following command several times,

with bolded parameters modified, I created the necessary Xen Unprivileged Domain (DomU) VMs:

$ xen-create-image –fs=ext3 –image=sparse –initrd=/boot/initrd.img-2.6.18-6-xen-686 –kernel =/boot/vmlinuz-2.6.18-6-xen-686 --modules= /lib/modules/2.6.18-6-xen-686/k --memory=256Mb –passwd –size=7Gb –swap=512Mb –dist=etch –mirror=http://ftp.fi.debian.org/debian/ --install-method=debootstrap –gateway=192.168.59.254 –ip=**192.168.59.10** –netmask=255.255.255.0 –mac=**00:16:3E:40:E8:FC** –dir=**/mnt/sdb3** –hostname=**Honeypot**

Once the Xen configuration file is created and DomU installations is complete, the Xen VM can be launched with the command:

$ xen create –c /etc/xen/Honeypot.cfg

In comparison to the VMware ESX server, the option of creating switches and doing similar networking setup in Xen is slightly more involved and requires a basic understanding of Linux networking. Although Xen is currently limited to having three switches only, this was sufficient for my setup.

3.4 Configuring Individual Honeynet Hosts

3.4.1 Low- and High-interaction HTTP Servers

First, I generated a self-signed SSL certificate to be used by the Apache HTTPS server, as shown in Listing 1.

```
$ openssl genrsa -out server.key 1024
$ openssl req -new -key server.key -out server.csr
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

Listing 1. Genrating SSL certificates

The Apache 1.3.23 and OpeSSL0.9.6c are compiled from source and installed on the Debian 3.1 host. The only needed change to enable HTTPS support in the Apache

server, once the *mod_ssl module* was set up, was to modify the SSLCertificateFile and SSLCertificateKeyFile declarations in httpd.conf file.

The HoneyWeb, emulating the LIH HTTP server, used the default configuration. Since the stunnel emulation of the HTTPS protocol on the LIH failed, I used the proxy option of Honeyd to redirect all HTTPS connections to HIH [35]. Listing 2 illustrates the configuration used for Honeyd.

```
create linux
set linux personality "Linux 2.3.12"
set linux default tcp action reset
add linux tcp port 80 "/usr/bin/python HoneyWeb-0.4.py Apache loopback.com 80 $ipsrc"
add linux tcp port 443 "proxy 192.168.59.102:443"
set linux uptime 3284460
bind 192.168.59.31 linux
```

Listing 2. Honeyd configuration file

The major obstacle for not using stunnel and running HTTPS locally on LIH is a bug in Honeyd, which prevents the Honeyd daemon from running as a privileged user and causes privileges to be dropped to the user ‘nobody’. When this occurs, the stunnel cannot get access to SSL keys and other related files that are not world-readable, thus failing to start the SSL daemon.

The Honeyd daemon was listening for incoming connections on a pseudo-interface and a corresponding static IP address created on the LIH host. Thus to make the Honeyd accessible to external clients, I added an explicit entry on the default gateway, specifying that the static Honeyd IP was accessible via the physical interface of the host running the LIH.

3.4.2 Honeynet Gateway Setup

The honeynet gateway has two functions: redirecting traffic between LIH and HIH honeypots and running Snort IDS with an empty rule set, which will be populated after new signatures are created. The traffic redirection is done using the tools provided by

the Honeybrid project [36]. The relevant extract of the Honeybrid configuration is shown in Listing 3.

```
target {
      filter "dst host 192.168.59.101 and port 443";
      frontend 192.168.59.101 "yesno";
      backend 192.168.59.102 "hash";
}
```

Listing 3. Honeybrid configuration extract

As can be seen from Listing 3, the Honeybrid filter rule has the Pcap-like syntax for accepting packets only for LIH honeypot, 192.168.59.101 on port 443. The frontend rule states that all the packets should be accepted for inspection by LIH, while the backend rule declares that only packets with a unique hash of payload should be redirected to the HIH honeypot. The hash rule was not effective in my setup, because all the traffic passing to the HIH honeypot was encrypted. Thus Honeybrid could see only unique encrypted packet payloads. However, I still use this rule, because of its efficiency for testing.

Overall, in the case of HTTPS protocol, the honeybrid operates in the following manner:

1. SYN packet is received by the Honeybrid and forwarded to both LIH and HIH honeypots.
2. Single SYN-ACK packet is sent in response to the external client.
3. The external client completes the TCP handshake by replying with ACK and sends a packet with ‘Client Hello’ as required by SSL/TLS protocol.
4. Honeybrid receives the ‘Client Hello’ packet and computes a hash on the TCP payload, since the ‘Client Hello’contains random bytes sent to an SSL server, the result is a unique packet, which is then forwarded to the HIH.
5. The RST packet is sent to the LIH to cause the LIH connection state to be cleared.
6. Further connections of the external client take place with the HIH only.

The resulting SSL session proceeded without interruption from the client point of view and the web page of the HIH honeypot was displayed to the external client, connecting to the honeynet with the address of the LIH.

The iptables configuration of the honeynet gateway, as required by the Honeybrid setup, consisted of two components: network address translation (NAT) and rules causing traffic to be directed to user space for processing and decision making. The complete iptables settings for Honeybrid are shown in Listing 4.

```
iptables -A PREROUTING -d 192.168.59.96/29 -j DNAT --to-destination 192.168.59.101
iptables -I FORWARD -d 192.168.59.101-j QUEUE //Incoming attack traffic
iptables -I FORWARD -s 192.168.59.101 -j QUEUE //Traffic outgoing from LIH
iptables -I FORWARD -s 192.168.59.102 -j QUEUE //Traffic outgoing from HIH
```

Listing 4. Firewall configuration of honeynet gateway

As presented in Listing 4, all traffic directed to 192.168.59.96 subnet was sent to the LIH host. The NAT rule acted to prevent external connections to be directly made to the HIH honeypot. The latter statements in Listing 4 indicate that both incoming and outgoing traffic was sent to Honeybrid application for inspection.

Snort IDS 2.8.5 is set up on the honeynet gateway. The main function of the Snort IDS in this project was to verify that a particular signature was functioning as expected. The Snort IDS can be located on any remote host and is placed on the honeynet gateway only for convenience. Listing 5 demonstrates parameters specified in snort.conf file.

```
var HOME_NET [192.168.59.0/32]
var EXTERNAL_NET !$HOME_NET

var HTTP_SERVERS any
var HTTP_PORTS [80,443]
var SHELLCODE_PORTS !80
var RULE_PATH /etc/snort/rules

preprocessor frag3_global:      max_frags 65536
preprocessor frag3_engine:      policy linux \
                                bind_to any \
                                detect_anomalies

preprocessor stream5_global:    track_tcp yes
preprocessor stream5_tcp:       policy linux, ports server
preprocessor stream5_udp:       ignore_any_rules

preprocessor http_inspect:      global \
                                iis_unicode_map unicode.map 1252

preprocessor http_inspect_server: server default  \
                                profile apache ports { 80 443 } \
                                oversize_dir_length 500 \
                                flow_depth -1

preprocessor ssl:               ports { 443 }

include $RULE_PATH/myrule.rules
```

Listing 5. Extract of Snort IDS configuration file

Listing 5 shows EXTRNAL_NET to be set for networks, other than the HOME_NET. Here I assumed that an attack on the honeynet would originate from networks, other than the ones where the honeynet resided. A fragmentation preprocessor, frag3, also needed to be configured as attack packets could be fragmented and pass undetected by existing Snort rules. Stream5 preporcessor was necessary to enable TCP and UDP connection tracking. Thus a direction of traffic flow could be specified in Snort rules. HTTP preprocessors were used to provide appropriate HTTP decoding and analysis for

the Snort IDS. The SSL preprocessor was used to detect attacks during SSL handshake phase, unfortunately SSLv2, exploited by the Linux Slapper worm, was not supported in the current Snort IDS release.

3.4.3 Setting Covert Monitoring

The compilation of Sebek client required installing Linux kernel headers and obtaining Linux kernel source code. The filter.txt of Sebek client was set to capture all traffic on port 443 and also all the keystokes executed by the user 'nobody', which is the user that is running an unprivileged Apache server and this is the identity obtained by an attacker once the OpenSSL vulnerability is exploited.

I compiled from source and installed Sebekd 3.0.3, the covert monitoring server, on the honeynet gateway. The Sebek server captured packets sent by the Sebek client, residing on the HIH honeypot. Although the honeywall developed by the Honeynet project provides a possibility for storing Sebek packets in the MySQL database, I chose to store Sebek server data in plain text files.

3.5 Verifying Honeynet Settings with Nmap

The honeynet scanning from anexternal network and detection of honeypots with Nmap is difficult, because of Honeybrid system not accepting packets containing bad checksums or a non-standard combination of TCP flags. Using the Nmap SYN scan with version detection I obtained the results shown in Listing 6.

```
$ nmap –sS –PN –n –vv –reason –A –T4 –p80,443 192.168.59.102 192.168.59.101
Interesting ports on 192.168.59.101:
PORT    STATE   SERVICE  REASON     VERSION
80/tcp   filtered  http          no-response
443/tcp   open    ssl/http   syn-ack Apache httpd 1.3.23 ((Unix) mod_ssl/2.8.6 OpenSSL/0.9.6c)
Interesting ports on 192.168.59.102:
PORT    STATE   SERVICE   REASON     VERSION
80/tcp   filtered   http          no-response
443/tcp   open     ssl/https?    syn-ack
```

Listing 6. Nmap scanning of Honeynet via Honeybrid

Because of filtering rules implemented by Honeybrid, all packets were accepted by Honeyd host, while only unique packets were directed to the HIH honeypot. This explains the incomplete version detection on port 443/tcp of 192.168.59.102, the HIH honeypot. Port 80/tcp appeared to be filtered, because the present Honeybrid 0.9 Beta version is limited to the support of only one port for replay and redirection.

During the next step I performed Nmap scanning without the Honeybrid present with the goal of detecting network-level differences between HIH and LIH honeypots. Using different TCP flags combinations for scanning and by sending packets with bad checksums I was not able to solicit a response that would distinguish HIH and LIH services. Listing 7 depicts the Nmap scans I attempted against the honeypots.

```
$ nmap –scanflags SRF -sV -A -T4 -n -vv -PN –p80,443 –send-ip 192.168.59.96/29
$ nmap -sS –badsum -sV -A -T4 -n -vv -PN –p80,443 –send-ip 192.168.59.96/29
```

Listing 7. Attempted honeypot detection with Nmap

3.6 Study of Apache Worm

3.6.1 Debugging Apache HTTP Server and SSL Plugin

OpenSSL debugging is a challenge because there is no process that is running an SSL library on its own. Instead OpenSSL is usually a part of an application using the SSL protocol. Fortunately, there are two standard utilities supplied by the library that can be used for debugging: s_client and s_server. Using s_server emulation of a basic SSL-enabled web-server is also possible. Listing 8 depicts a setup of a sample SSLv2 session alongside with the output of s_client.

```
$openssl s_server -accept 443 -cert server.crt -key server.key -debug -state -msg -ssl2 -WWW
$ openssl s_client -connect 192.168.59.102:443 -msg -ssl2
>>> SSL 2.0 [length 002b], CLIENT-HELLO
<<< SSL 2.0 [length 0365], SERVER-HELLO
>>> SSL 2.0 [length 0092], CLIENT-MASTER-KEY
>>> SSL 2.0 [length 0011], CLIENT-FINISHED
<<< SSL 2.0 [length 0011], SERVER-VERIFY
<<< SSL 2.0 [length 0011], SERVER-FINISHED
SSL-Session:
   Protocol  : SSLv2
   Cipher    : DES-CBC3-MD5
   Session-ID: 881FA7D91....
```

Listing 8. SSLv2 session handshake

As can be seen from Listing 8, initially a negotiation of cipher suites between the client and the server takes place during client and server hello messages. Sending of the CLIENT-MASTER-KEY serves establishing a common session key between a client and server to be used for encryption and decryption.

The Apache HTTP server debugging is challenging with Apache 1.3.23 version, because running Apache in a single mode is not available. I was able to compile Apache with debugging symbols successfully and execute the HTTP daemon with Valgrind's massif, the heap profiler. An extract of the results obtained from running the heap profiler during normal HTTPS session is shown in Listing 9; the heap snapshot belongs to the Apache HTTPS child process.

```
$valgrind –tool=massif ./httpd -f /usr/local/apache/conf/httpd.conf –DSSL
Heap allocation functions accounted for 81.6% of measured spacetime
Called from:
  31.0% : 0x80E0F98: CRYPTO_malloc (mem.c:215)
  16.1% : 0x807C6AB: malloc_block (alloc.c:253)
   9.5% : 0x80A318A: ap_hook_create (in /root/apache_1.3.23/src/httpd)
   5.5% : 0x80E102F: CRYPTO_realloc (mem.c:231)
```

Listing 9. Heap allocation profile of Apache HTTPS daemon

As seen from Listing 9, the heap is being actively used by the HTTPS daemon, with CRYPTO_malloc function of the OpenSSL library occupying the largest portion of the heap. The CRYPTO_malloc call is a wrapper for Glibc's malloc function. Executing an exploit of theHTTPS daemon running with Valgrind heap profiler failed. Therefore my observed results of the heap allocation during exploitation were the same as for a non-exploited HTTPS session.

One of the important features that allow the shellcode execution to succeed is memory allocation process, performed by the Apache web-server. Being a process-oriented server and with the server spool exhausted, Apache 1.3.23 HTTPS server allocates memory for the vulnerable SSL_SESSION structure in the same memory region for two consecutive SSL connections, thus allowing an attacker to reuse the offsets leaked during the first handshake.

I wanted to verify that memory addresses for SSL_SESSION structure are indeed the same for two independent, consecutive connections. Thus I modified modules/ssl/ssl_engine_kernel.c file of mod_ssl module, as shown in Listing 10, to print out the memory address each time a block for SSL_SESSION was allocated.

```
/* This callback function is executed by OpenSSL whenever a new SSL_SESSION is
added to the internal OpenSSL session cache. */
int ssl_callback_NewSessionCacheEntry(SSL *ssl, SSL_SESSION *pNew)
{
…
                        ssl_log(s,SSL_LOG_INFO,"SSL_SESSION=%d",pNew);
```

Listing 10. Modified SSL_SESSION handler

After recompiling Apache and mod_ssl server I was able to see that the connections established to the HTTPS server had the SSL_SESSION structure situated in the same memory address.

3.6.2 Understanding Shellcode Used

For simplicity I took only a short extract of the shellcode present in the exploit. In the case of the Linux Slapper worm, the execution of setresuid() call is used for obtaining root privileges for the bash shell, although unsuccessfully since Apache is usually running as an unprivileged process. This brief study provides a good illustration of the methods used by shellcode writers.

The default code of a setrestuid() call that is generated by GCC 4.3.3 has the structure shown in Listing 11.

```
8048241:   c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
8048249:   c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
8048251:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
8048258:   e8 c3 64 00 00          call   804e720 <__setresuid>
```

Listing 11. A setresuid() call Assembly and hexdump

The Assembly code in Listing 11 shows three zeros pushed to the stack, before the setresuid() subroutine is called. However, such code is not suitable for injecting into vulnerable buffers, because of zeros used to terminate strings and contained in the hex dump of this code. Knowing how a system call is made in Linux: load EAX register with a syscall number, use EBX, ECX and EDX to pass arguments for a syscall and issue int 0x80 software interrupt to switch from protected level 3 to kernel level 0, and a code shown in Listing 12 is produced.

```
80483a5:   bb 00 00 00 00    mov   $0x0,%ebx
80483aa:   b9 00 00 00 00    mov   $0x0,%ecx
80483af:   ba 00 00 00 00    mov   $0x0,%edx
80483b4:   b8 a4 00 00 00    mov   $0xa4,%eax
80483b9:   cd 80             int   $0x80
```

Listing 12. Handwritten Assembly setresuid() syscall

The code in Listing 12 is more straightforward, but has the same issue with NULL bytes, as extract in Listing 11. The zeros of the MOV to EAX instruction occur from the system being 32-bit, while only an 8-bit move is performed. By addressing only lower bytes of the EAX register, the unneeded zeros are removed. A well-known workaround for initializing registers with zeros is to use the XOR instruction leading to the final version presented in Listing 13.

```
80483a5:      31 db        xor    %ebx,%ebx
 80483a7:     31 c9        xor    %ecx,%ecx
 80483a9:     31 d2        xor    %edx,%edx
 80483ab:     b0 a4        mov    $0xa4,%al
 80483b0:     cd 80        int    $0x80
```

Listing 13. Version of shellcode without zeros in the hex dump

Listing 14 is a shellcode version of the same setresuid() syscall that was used by a worm itself.

```
31 c9      xor  %ecx,%ecx
f7 e1      mul  %ecx,%eax
51         push  %ecx
5b         pop  %ebx
b0 a4      mov  $0xa4,%al
cd 80      int   $0x80
```

Listing 14. Shellcode used by the worm

As shown in Listing 10, first, the ECX is cleared, then a multiplication of two 32-bit numbers takes place, thus the high-order bits of the result are placed in EDX, while the low-order bits are stored in EAX, and thus both registers contain zeros. PUSH and POP are executed to avoid zeros in the hex dump and initialize the EBX. Finally, a syscall interrupt takes place.

3.6.3 Preparing Exploit Code

For the purposes of testing my honeynet I did not use the actual Linux slapper worm, but instead the exploit code that was made specifically to perform the targeted exploits of Apache HTTPS servers. Thus I was able to verify a successful worm infection much more quickly, than in the case of using the sequential scanning worm, and without a risk

of the worm causing damage to external networks. The exploit code mimics the behaviour of a worm, except for the creation of a control Peer-to-Peer (P2P) network.

The worm is working by exploiting heap overflow of the OpenSSL library before 0.9.7e that allows writing to any memory location with four bytes of arbitrary data. To make the exploit work, there also needs to be an appropriate version of the Glibc library installed. However, this requires running Debian 3.1, because later releases of Debian rely on the updated Glibc library patched against attacks on the unlink macro, and there are no easily available mechanisms for the rollback of the Glibc package.

Understanding the exploit code requires a basic knowledge of the Executable and Linking Format (ELF) format. Every ELF file consists of sections. For example .text section is used for storing the read-only program code in the memory, while .data section contains the variables initialized in the program. There is also a .got section, standing for Global Offset Table (GOT) providing reference to external shared libraries where most of the basic functions reside. By making use of the heap overflow and unlink macro feature of Glibc 2.3.2, the exploit is overwriting the initial address of free() function in the GOT table of the httpd ELF binary, and therefore the control flow is redirected to the shellcode.

For the exploit to work, it needs to know the address of thefree() function in the httpd binary. Listing 15 shows how the GOT section and the address of free() syscall are obtained.

```
$ objdump -R /usr/local/apache/bin/httpd | grep free
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE             VALUE
…
08177598 R_386_JUMP_SLOT   free
```

Listing 15. GOT table and free() address of httpd binary

The command in Listing 15 is used to display the dynamic relocation entries of a file. The address of the GOT table is stored in the EBX register; thus the number produced

by the objdump command represents the offset into the GOT to reach a certain function. Once this offset is found, it will simply be added to the array of the known offsets in the exploit.

3.6.4 Web-server Exploitation Technique

For executing the worm I used the code found on an Internet website of bugtraq.c file. The source code for this worm has become available because of the error made by the author: the malicious code compiling and executing a worm instance on an infected host did not delete the source code after compilation. The particular worm sample I used contained all the original elements of P2P networking and scanning for vulnerable hosts. However, upon a discovery of a vulnerable host, the program simply terminates without exploitation taking place.

The User Datagram Protocol (UDP) P2P network is using port 2002 on an infected computer, providing a backdoor into the compromised system. The UDP packets are not being encrypted, but the reliability of the UDP connection is being ensured by the use of checksums and acknowledgments. Using the worm client program it is possible to direct Denial of Service (DoS) attacks on other hosts by creating UDP or TCP floods of requests. I verified this functionality, although with a small number of hosts there can be no DoS attack, as each infected host sends one connection request.

Interestingly, although there is potential for a worm to coordinate scanning with other worm instances, based on the overlay UDP network, the worm instances do not appear to synchronize scanning results. The scanning is based on randomly chosen targets, using the set of precompiled first octets, random selection of the second octet and scanning the remaining octets sequentially.

During my experiments with the worm, I was not able to perform targeted infection of a particular host, although properties of several worm commands remain obscure to me. The UDP packets exchanged by UDP peers do not contain American Standard Code for

Information Interchange (ASCII) characters, complicating the creation of signatures for this worm, without understanding worm internals. In the laboratory environment I launched the worm and observed scanning being performed. The average rate, SYN packets, of worm scanning for potential victims is 32 connections per second.

3.6.5 Heap Overflow – Low Level Perspective

The Linux Slapper worm used a two-stage approach for attacking a victim HTTPS server. During the first stage, by means of a heap overflow, the SSL_SESSION structure information is being overwritten, causing the server to leak the values of internal SSL-related structures of the server. During the second stage, another heap overflow takes place and the previously leaked server-side SSL structures are reinserted into the server. Additionally, by using the leaked addresses an attacker obtains a reliable reference for shellcode location. Finally, an attack on Glibc takes place, modifying internal heap structures and causing the GOT entry of free() syscall being overwritten with an address of the shellcode. Once this is accomplished, a client asks a server to free the SSL_SESSION structure, by closing the connection, and the execution control will be transferred to the shellcode supplied by an attacker. [37]

On the network level, the first heap overflow is caused by a Client Master Key packet with a key argument field employed for overflowing the vulnerable server-side buffer. Figure 4 illustrates the malicious packet sent by a client.

```
00b0  8a 3b 18 c9 76 c3 6f 74  eb 05 a3 67 0b 45 52 de   .;..v.ot ...g.ER.
00c0  fb be 56 57 8e 2c 28 9d  fa e0 c0 09 22 3c 67 22   ..VW.,(. ...."<g"
00d0  7a 27 6f 31 13 4c 41 41  41 41 41 41 41 41 41 41   z'o1.LAA AAAAAAAA
00e0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
00f0  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0100  41 41 41 41 41 41 41 41  41 41 70 00 00 00         AAAAAAAA AAp...
```

Figure 4. Hex dump extract of the first overflow packet

As can be seen in Figure 4, initially a padding of 52B consisting of the ‘A’ symbol is inserted to represent the Client Master Key, while the last four-bytes 0x70 or 112B set the new size of session information length, causing additional and normally hidden data

to be leaked to a client. Having obtained the needed offsets, an attacker creates a second SSL connection, reinserting previously leaked values and injecting the shellcode. Figure 5 shows the contents of the second packet, also sent as Client Master Key.

```
0130  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0140  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41   AAAAAAAA AAAAAAAA
0150  41 41 41 41 41 41 00 00  00 00 00 00 00 00 41 41   AAAAAA.. ......AA
0160  41 41 01 00 00 00 41 41  41 41 41 41 41 41 41 41   AA....AA AAAAAAAA
0170  41 41 68 d1 16 08 41 41  41 41 00 00 00 00 00 00   AAh...AA AA......
0180  00 00 00 00 00 00 41 41  41 41 41 41 41 41 00 00   ......AA AAAAAA..
0190  00 00 11 00 00 00 8c 75  17 08 50 88 1d 08 10 00   .......u ..P.....
01a0  00 00 10 00 00 00 eb 0a  90 90 90 90 90 90 90 90   ........ ........
01b0  90 90 31 db 89 e7 8d 77  10 89 77 04 8d 4f 20 89   ..1....w ..w..O .
01c0  4f 08 b3 10 89 19 31 c9  b1 ff 89 0f 51 31 c0 b0   O.....1. ....Q1..
01d0  66 b3 07 89 f9 cd 80 59  31 db 39 d8 75 0a 66 b8   f......Y 1.9.u.f.
01e0  99 d0 66 39 46 02 74 02  e2 e0 89 cb 31 c9 b1 03   ..f9F.t. ....1...
01f0  31 c0 b0 3f 49 cd 80 41  e2 f6 31 c9 f7 e1 51 5b   1..?I..A ..1...Q[
0200  b0 a4 cd 80 31 c0 50 68  2f 2f 73 68 68 2f 62 69   ....1.Ph //shh/bi
0210  6e 89 e3 50 53 89 e1 99  b0 0b cd 80               n..PS... ....
```

Figure 5. Hex dump extract of the second packet

In Figure 5, the 0x0816D168, underlined in green, is the leaked address of server-side SSL structure during the first overflow. The 0x0817758C, underlined in red, is the address of GOT entry of free() syscall minus 12B, which is obtained by fingerprinting a vulnerable server. The 0x081D8850 is an address where the shellcode is located, and this is the value with which the GOT entry of free() is overwritten. The 0x081D8850 should be obtained from the second leaked value from the server-side SSL structure, but the second leaked value is not seen in this packet.

Once the exploit has succeeded, the shellcode will locate the necessary file descriptor of the attacker's connection on the server-side. Finally, the attacker spawns a shell, reusing the connection established by the exploit, with the packet sending done in clear text. The bash shell spawning packet is shown in Figure 6.

```
)030  01 db e8 28 00 00 01 01  08 0a 00 05 b7 87 00 02   ...(.... ........
)040  bf 6a 54 45 52 4d 3d 78  74 65 72 6d 3b 20 65 78   .jTERM=x term; ex
)050  70 6f 72 74 20 54 45 52  4d 3d 78 74 65 72 6d 3b   port TER M=xterm;
)060  20 65 78 65 63 20 62 61  73 68 20 2d 69 0a 0a       exec ba sh -i..
```

Figure 6. Spawning Bash shell on exploited host

A result of the packet in Figure 6 sent by an attacker is a shell for the user nobody, a user that usually has the privilege of running the Apache server.

3.7 IDS Signature Generation

The overall process of extracting signatures from traffic directed to the HIH honeypot and captured by the honeynet gateway consists of the following steps:

1. extract keystrokes logs from Sebek-server daemon running on honeynet gateway
2. parse relevant traffic capture and locate packets corresponding to keystrokes executed on HIH honeypot
3. place the payload hex dump of packets containing Sebek keystrokes into an IDS signature.

The script for producing signatures, included in Appendix 1, is intended to be run after the necessary Sebek logs and Pcap files have been collected. The Sebek log parsing consists of extracting the keystrokes logged and constructing an ASCII string that can be later looked up from the Pcap files. Next, assuming that every command is separated by a semicolon, I proceeded to look for substrings of the complete keystrokes' string within the Pcap file to obtain timestamps of relevant packets. Once this was accomplished, I used the Pcap file timestamps to retrieve the corresponding packets and extract the hex dump for signatures. This two-step approach was needed, because of the parsing process being implemented without knowing how to extract proper fields of a TCP packet using Perl. This was an inflexible solution, but served as a proof of the concept.

3.7.1 Capturing Traffic Traces

The tcpdump tool is capturing packets directed to the HIH honeypot, although the Net::Pcap library can be used for scripting the process of capturing. The command line is used for simplicity:

```
$ tcpdump -i eth0 -s 1500 -w honeynet_capture.pcap host 192.168.59.70 and tcp port 443,
```

where 192.168.59.102 is the address of HIH honeypot.

Once the capturing is completed, there will be a need to eliminate any unneeded packets, for example TCP control packets or encrypted sessions from the captured file to allow the resulting hex dump to be placed into a signature. Eliminating TCP control packets can be done using a statement such as:

```
$ tcpdump -p -Xnr  honeynet_capture.pcap 'tcp[tcpflags] & (tcp-syn|tcp-fin|tcp-ack) == 0' |
tcdump -p -r - -w no_tcp_control_pkts.pcap
```

Later the result can be parsed with ssldump, used to identify SSL sessions and print them out:

```
$ ssldump -Xnr no_tcp_contr.pcap
```

However, this solution was not suitable for my purposes, because it produced encrypted packets only, while I needed the unencrypted hex dump. Because there was no tool to compute the difference of two Pcap files, I had to use another approach, this time with the tshark tool. I relied on the feature of tshark to accept the same filtering parameters as Wireshark. Thus if I needed to look for an SSL session and print hex dump, I would state:

```
$ tshark -xR ssl -r  honeynet_capture.pcap
```

By manipulating the output of the tshark command I was able to obtain the necessary hex dump to be inserted into an IDS signature.

3.7.2 Existing Signature in Snort IDS Ruleset

An interesting aspect of the Apache SSL exploit allowing to make effective network IDS signatures is that the exploit code is being transmitted in clear text before a secure connection is established. The current Snort IDS rule aimed at capturing the Linux Slapper exploit contains the statement shown in Listing 16.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 443 (msg:"MISC OpenSSL
Worm traffic"; flow:to_server,established; content:"TERM=xterm"; nocase; reference:url
,www.cert.org/advisories/CA-2002-27.html; classtype:web-application-attack; sid:1887;
rev:3;)
```

Listing 16. Snort IDS rule for Linux Slapper worm

The Snort rule, shown in Listing 16, relies on detecting “TERM=xterm” string within the packet payload. During my experiments I confirmed that this rule was effective, as this packet occurred immediately after the successful exploit. I also measured the processing time of the original Snort IDS signature, using a simple Perl script, shown in Appendix 2, to record time with nanosecond precision. The time needed for Snort IDS to successfully process the Pcap file and detect an attack, using only the Linux Slapper worm signature was 0.389992 seconds, based on an average of 20 runs.

4 Results

During my experiments I verified that Apache 1.3.23 with mod_ssl 2.8.6 and OpenSSL 0.9.6c, running on Debian 3.1 with Glibc 2.3.2 was vulnerable to Linux Slapper worm infection. A successful exploitation provided an attacker with a shell access of the user 'nobody' to an affected host. Such privileges were sufficient for worm propagation and creation of the overlay P2P network, although sensitive information such as SSL private keys were not compromised.

The output of a signature generation script showed only the byte sequence to be added to a Snort IDS signature. A complete version of Snort IDS signature, based on the script output, is shown in Listing 17.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 443 (msg:"Attack detected";
flow:to_server,established;
content:"|5445524d3d787465726d3b20657865632062617368202d690a0a|";
sid:1000010;)
```

Listing 17. Snort IDS signature produced by honeynet

In Listing 17, all elements of the signature, except for the byte sequence, were a signature template and existed across all signatures. The time of Snort IDS processing this signature was 0.392276 seconds, compared with the original Snort signature time of 0.389992 seconds. The produced signature was almost a hexadecimal representation of ASCII keystrokes, thus the usefulness of the result is minimized. Based on my algorithm for signature generation I also obtained two extra byte sequences for other parts of keystrokes string. However, I did not use those strings in my performance evaluation.

The redirection process provided by Honeybrid is not effective, because of the majority of exploit traffic being encrypted, meaning that all the connection load is handled by the HIH honeypot and is not shared across LIH and HIH honeypots. On the other hand, redirection performed by Honeybrid is transparent and cannot be noticed by an attacker. Unfortunately, the present release of Honeybrid is limited to supporting only one LIH

and one HIH honeypot; thus scalability of this solution can be an issue. The Sebek covert data collection was successful and provided sufficient information on the attacker's activity, although, due to difficulties with Sebek compilation, for platforms other than Debian 3.1 and Debian 4.0, the application of Sebek service was limited.

I also observed that without the signature an attack would pass undetected by Snort IDS. This situation was improved with the introduction of the SSL pre-processor in the Snort IDS, although the exploit operates with SSLv2, that is not supported by the current release of Snort IDS. An alternative to making an IDS application-aware is having honeypots or end-user systems to supply IDS context information and thus generic detection solutions can be developed.

At the moment considerably more effort is spent on developing an automated signature generation algorithm. The attractiveness of obtaining such a solution is minimized, if an exploit analyst can understand the workings of a network protocol and use the relatively simple syntax of Snort IDS rules to create new policies based on a sample of malicious traffic. However, such procedure can only be performed after the vulnerable hosts are infected and honeypots can provide a certain time benefit for reacting to new threats.

5 Discussion

The signature algorithm that I chose to use in this project is restricted to the Apache SSL exploit and does not provide a generic solution for making NIDS signatures from network traffic. While knowledge of exploit internal functioning is required for making an IDS signature, this also means that an algorithm for signature generation is customized to suit the needs of a particular exploit. It is also not practical to obtain a single solution for all network protocols, but by analyzing more exploits of a specific protocol, a more generic and robust solution could be implemented.

During my work I dealt with a worm, that neither actively attempted to hide itself, nor used polymorphism to avoid being detected by NIDS systems. Fortunately, the worm had exploited a heap overflow of a victim host, using SSL protocol, and did not attempt to use encryption for communication at a later stage.

The implementation of the honeynet system showed that the information about activity on HIH honeypots, in response to traffic seen by NIDS is valuable. Without such information an attack can be detected only by IDS knowing the protocol structure and constraints, monitoring inconsistencies in the connection state or observing the connection rate. Using IDS to simply match traffic patterns is not an efficient method of using IDS resources, as the functionality provided by an IDS is more granular. The detection can also be based on a string offset inside a packet or TCP flags and options used by an attacker. Such information is readily available to my honeynet system. However, I did not implement these options.

The honeywall system did not prove useful in the honeynet, because of my concentration on a non-self-propagating code. Additionally, the command line interface for retrieving tcpdumps was not available, which significantly reduced the benefits of running a honeywall in my setup. I was not able to use honeywall with Xen either,

because of the needed modification to the kernel and the need to recreate all software packages and configurations manually after kernel modifications.

While LIH honeypots are able to emulate only a small portion of the properties available on a HIH HTTP server, it is sufficient for an automated system, because scanners used in present worms do not usually distinguish minor fingerprint differences. With this honeynet system not targeting a determined human attacker, such an approach should be sufficient.

The study showed that a more versatile interface for reading Pcap files is also required. In particular the functionality that is being already implemented with Wireshark dissectors in GUI, such as identifying different sections of payload and protocol stack, needs to be transferred to console applications such as tshark. One of the issues causing my signature-generating solution to be non-generic, is the reliance on fixed offsets into a packet where the needed information resides. By being able to extract an exact portion of the packet payload, such as SSLv2 Client Hello message hexdump, there is a possibility to produce more efficient signatures.

Ideally, a honeynet should not only attract an attacker, but also provide means for analyzing malicious activity by collecting malware samples being uploaded to honeypots. In a fully automated system the collected malware samples should later be executed and monitored. One of the challenges of such design is establishing a secure communication channel from HIH honeypots to the analysis server, since an analysis of malware samples locally on the HIH creates a risk of the HIH being detected.

For the purposes of covert monitoring, there is more development potential in working with open-source hypervisors, then with closed-source ones. One advantage is a prospect to modify hypervisor source code to implement interception of guest VM system calls in the virtualization layer. Not only will such monitoring be more covert, but also provide methods for dumping affected memory regions and sharing complete status information with the malware analysis service. An implementation of the external

guest VM management system in the case of VMware ESX server requires building additional infrastructure that imposes an unnecessary overhead for a small-scale project. However, in case of Xen the needed interfaces are readily available and require less configuration effort.

Another difficulty for automated honeynet development is detecting a moment when a honeypot is infected and needs to be removed from a honeynet for re-initializing. This is a particularly sensitive area for automation. A person monitoring a honeynet will be able to identify when an attack is over or no additional information is required. However, making such a decision by honeynet itself requires either implementing certain heuristics or reliance on a fixed time interval. A need for maintaining long-term images of honeypots may arise during multiple stage attacks when a worm is updating a copy of itself or downloading additional attack tools.

While choosing a particular exploit to use for this project I was surprised by the complexity and multitude of conditions that needed to be satisfied for creating a vulnerable setup. Establishing such honeypot environments requires relevant knowledge of the particular application and plugins, as is the case with recent Apache, Tomcat and cross-site scripting exploits. Additionally, implementing LIH honeypots by writing service scripts for such systems is not practical. One of the insights into this challenge is creating LIH honeypots from a Pcap traffic capture of a particular protocol.

Exploit writing, IDS systems and honeypots require an intersection of skills not only in computer networks, but also largely in software development. Because the goal of a defence tool, such as honeynet, is to provide security proactively, by knowing about probable errors in the software development process, a defence system can be targeted more accurately and have improved detection rates. Moreover, the choice of protocols to be used on HIH honeypots needs to be consistent and reflect the security needs of an organization.

6 Conclusion

In this project a semi-automated system for identifying malware samples for the purposes of creating network IDS signatures was implemented. Additionally, a study of the Apache HTTPS worm was undertaken. Using the knowledge of the Apache HTTPS worm techniques, I created IDS signatures, based on the honeynet system, providing reliable application layer information to inform about the threat of the traffic seen by the IDS as an encrypted SSL session. Thus the goals of the project were achieved.

Although, the developed system is not suitable for use in production environments, my experiment provides an insight into the operation of defence and attack methods. The produced results also identify a lack of effective covert monitoring embedded into the virtualization layer and ineffectiveness of command-line traffic analysis tools to provide an interface for accessing the packet structure. The major disadvantage of this study is concentration on a single case-specific exploit and therefore the selected method has only limited application potential.

Recommendations for further study may include a closer look at the internal functioning of the CPU and corresponding virtualization implications, as well as an examination of malware classes and formulation of the formal rules for worm propagation. Such a study could result in the formulation of novel models in malware detection, because fewer layers of abstraction can help in finding more efficient and obvious similarities to be used for IDS signatures.

References

1. Peter Mell, Karen Kent, Joseph Nusbaum. Guide to Malware Incident Prevention and Handling, Computer Security Division Information Technology Laboratory National Institute of Standards and Technology; November 2005

2. Yi Tang, Xiangning Dong. Anting: An Adaptive Scanning Method for Computer Worms, Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence

3. Chuanyi Ji, Zesheng Chen. A Self-Learning Worm Using Importance Scanning, WORM'05, Fairfax, Virginia, USA; November 11, 2005

4. Microsoft Honeymonkey project [online]. URL: http://research.microsoft.com/ en-us/um/redmond/projects/strider/honeymonkey Accessed 17 April 2010.

5. Lok Kwong Yan. Virtual honeynets revisited, Proceedings of the 2005 IEEE Wokrshop on Information Assurance and Security, United States Military Academy, West Point, NY

6. J. Briffaut, J.-F. Lalande, C. Toinard. Security and Results of a Large-Scale High-Interaction Honeypot , Journal of computers 2009; 4(5)

7. Hassan Artail, Haidar Safa, Malek Sraj. A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks, Computers and security 2006; 25: 274 – 288

8. Xuxian Jianga, Dongyan Xua, Yi-MinWangb. Collapsar:AVM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention, Journal of parallel and distributed computing 2009; 66: 1165 – 1180

9. Michael Vrable, Justin Ma, Jay Chen, David Moore. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm, Collaborative Center for Internet Epidemiology and Defenses Department of Computer Science and Engineering University of California, San Diego; 2005

10. Paul Barham, Boris Dragovic, Keir Fraser. Xen and the Art of Virtualization, Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), University of Cambridge Computer Laboratory, UK; October 2003

11. John Scott Robin, Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, USA; August 14 –17, 2000

12. Joanna Rutkowska. Security Challenges in Virtualized Environments, Invisible Things Lab, Nordic Virtualization Forum; October 2007, URL: http://invisiblethings.org/papers/Security%20Challenges%20in%20Virtualized%20Enviroments.pdf Accessed 17 April 2010

13. Srilatha Chebrolua, Ajith Abrahama,b, Johnson P. Thomas. Feature deduction and ensemble design of intrusion detection systems, Computers & Security 2004; 24: 295 – 307

14. Eugene H. Spafford, Diego Zamboni. Intrusion detection using autonomous agents, Computer Networks 2000; 34: 547 – 570

15. Performance Rules Creation [online]. Writing Effective Rules, Part I, URL: http://assets.sourcefire.com/snort/vrtwhitepapers/performance_rules_creation_1.pdf Accessed 17 April 2010

16. Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code , WORM'03, Washington, DC, USA; October 27, 2003

17. Holger Dreger, Christian Kreibich , Vern Paxson. Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context , Congrès DIMVA 2005 :detection of intrusions and malware, and vulnerability assessment, Vienna; July 7 - 8 2005

18. Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis. Towards Collaborative Security and P2P Intrusion Detection, Proceedings of the 2005 IEEE Workshop OD Information Assurance and Security, United States Military Academy

19. Tal Garfinkel, Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection, Computer Science Department, Stanford University; 2003

20. Andreas Moser, Christopher Kruegel, Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis, 2007 IEEE Symposium on Security and Privacy

21. Carsten Willems, Thorsten Holz, Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox, IEEE Security and Privacy 2007: 32 – 29

22. David Moore, Colleen Shannon, Geoffrey M. Voelker. Internet Quarantine: Requirements for Containing Self-Propagating Code, IEEE Proceedings of the INFOCOM 2003.

23. Hyang-Ah Kim, Brad Karp. Autograph: Toward Automated, DistributedWorm Signature Detection, Proceedings of the 13th USENIX Security Symposium; August 9–13, 2004

24. Sumeet Singh, Cristian Estan, George Varghese. AutomatedWorm Fingerprinting, OSDI ’04: 6th Symposium on Operating Systems Design and Implementation, USENIX

25. Christian Kreibich, Jon Crowcroft. Honeycomb. Creating Intrusion Detection Signatures Using Honeypots, University of Cambridge Computer Laboratory, UK; 2nd Workshop on Hot Topics in Networks (HotNets-II), Boston, USA; 2003

26. Jianwei Zhuge, Thorsten Holz, Xinhui Han. Collecting Autonomous Spreading Malware Using High-Interaction Honeypots, ICICS 2007, LNCS 4861: 438–451; 2007

27. Jamie Twycross, Matthew M. Williamson. Implementing and testing a virus throttle, Proceedings 12th USENIX Security Symposium, Washington, DC, USA; August 4 – 8 2003

28. Gordon “Fyodor” Lyon. Nmap network scanning: official Nmap project guide to network discovery and security scanning, Insecure.Com LLC; 2008.

29. Niels Provos, Thorsten Holz. Virtual Honeypots: From Botnet Tracking to Intrusion Detection, Addison-Wesley Professional; 2007

30. Thorsten Holz, Frederic Raynal. Detecting Honeypots and other suspicious environments, United States Military Academy, Proceedings of the 6th IEEE Information Assurance Workshop, IEEE; 15 June 2005

31. Ken Kato. VMware backdoor commands [online] URL: http://chitchat.at.infoseek.co.jp/vmware/backdoor.html Accessed 17 April 2010.

32. Thorsten Holz, Maximillian Dornseif, Christian N. Klei. NoSEBrEaK - Attacking Honeynets, United States Military Academy, Proceedings of the 5th Annual IEEE Information Assurance Workshop, West Point; 11 June 2004

33. Jungsuk Song, Hiroki Takakura. Cooperation of Intelligent honeypots to detect Unknown malicious codes, Information Security Threats Data Collection and Sharing, 2008. WISTDCS '08: 31 - 39

34. Sherif M. Khattab, Chatree Sangpachatanaruk, Daniel Mosse. Roaming Honeypots for Mitigating Service-level Denial-of-Service Attacks, Proceedings

of the 24th International Conference on Distributed Computing Systems, IEEE; 2004

35. Kevin Timm. HoneyWeb-0.4 scripts for Honeyd [online], URL: http://www.honeyd.org/contrib.php Accessed 17 April 2010.

36. Robin Berthier, Thomas Coquelin, Julien Vehent. Honeybrid project, URL: http://honeybrid.sourceforge.net Accessed 17 April 2010

37. Frédéric Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit [online], Symantec Security Response, URL: http://www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf Accessed 17 April 2010.

Appendix 1: Script for Extracting Hex Dumps from Pcap File

```
#!/usr/bin/perl

my $pcapfile="capture.pcap";
my $sebek_log="sebek_log.txt";

my %header;
my @res=();
my $string;
my $bytes;

# Parse the Sebek log here
# Assuming Sebek has the form, ex:
# [1269619236.199631  type=(sys_read) ip=(192.168.59.102) pid=(741:776) command=(sh)
#uid=(65534) inode=(0) fd=(0) len=(1)]e

{ my $cmd="cat $sebek_log | grep \"len=(1)\" | sed 's/.*\]//'";
my @out=`$cmd`;
foreach my $char (@out) {
            $string .= $char;
            chomp($string);
}
print $string . "\n"; }

# Process semicolon separated fields of the string one by one
# Assuming the attack string is similar to:
# TERM=xterm;exportTERM=xterm;...

my $i=1;
while ($i < 9) {
    my $cmd="echo -n \'$string\' | awk -F \";\" \'{print \$$i}\'";
    my @out=`$cmd`;
    chomp(@out);

    # Ask tshark to only show packets, which payload contains one of
    # the words from the attack string

    my $tshark="tshark -x -r $pcapfile -t e -R \'tcp contains \"@out\"\'";
    my @tout=`$tshark`;

    # Get the timestamps of the suspicious packets
    foreach my $line (@tout) {
        if ($line =~ m/TCP/) {
            my $cmd="echo -n \'$line\' | awk \'{print \$2}\'";
            my $output=`$cmd`;
                                    push @res,$output;
            print $output;
        }
    }
    $i++;
}
```

Appendix 1: Script for Extracting Hex Dumps from Pcap File

```
foreach my $time (@res) {
		chomp($time);
		my $t="tshark -x -r $pcapfile -t e | grep -A 9 $time | grep -v $time | sed 's/[0-9][0-9][0-9][0-9]\\s\\s//' | awk -F \" \" '{\$17=\"\"; print}' | awk -F \" \" '{\$17=\"\"; print}' | sed 's/exec//' | sed 's/\\s//g'";
		my @out=`$t`;
		foreach my $l (@out) {
				$bytes .= $l;
				chomp($bytes);
		}
		# Remove Ethernet + IP + TCP headers = 66B or 132 characters
		my $sign=substr($bytes,132);
		print "Signature:" . $sign . "\n";
```

Appendix 2: Script for Measuring Snort IDS Signature Execution Time

```
#!/usr/bin/perl

# Snort command to use
my $snort="snort -d -l /etc/snort/log -c /etc/snort/snort.conf -r ~/perl_test/new/capture.pcap";
my $date="date +%s.%N";

# Save output to a log file
open(LOG,">>snort_time.log");

my $t1=`$date`;
my @out=`$snort`;
my $t2=`$date`;
my $diff=$t2-$t1;

# Report the time difference
print LOG "$diff\n";
close LOG;
```