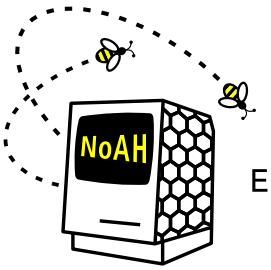
SIXTH FRAMEWORK PROGRAMME

Structuring the European Research Area Specific Programme

RESEARCH INFRASTRUCTURES ACTION



European Network of Affined Honeypots

Contract No. RIDS-011923

D1.4 Architecture Integration

Abstract:

This document provides an overview of all NoAH components, defines their requirements and describes the interface between them. The NoAH architecture, as described so far, is a set of individual components that cooperate to form a farm of distributed honeypots. Although the main NoAH components –low- and high-interaction honeypots, signature generation, honey@home and funneling/tunneling – can work individually, they need a well-defined interface between them in order to cooperate. In this deliverable we set the interfaces between honey@home and the NoAH core, low-interaction honeypots with high-interaction ones and finally high-interaction honeypots with signature generation mechanisms. Furthermore, besides the traffic and detection components, a complete architecture requires a database that will store meta-data about honeypot status, data collected and operational roles. We describe the design of the database also in this document.

Contractual Date of Delivery	M18
Actual Date of Delivery	M19
Deliverable Security Class	Public

The NoAH Consortium consists of:

FORTH	Coordinator	Greece
VU	Principal Contractor	The Netherlands
TERENA	Principal Contractor	The Netherlands
FORTHnet	Principal Contractor	Greece
DFN-CERT	Principal Contractor	Germany
ETHZ	Principal Contractor	Switzerland
VTRIP	Principal Contractor	Greece
ALCATEL	Principal Contractor	France

Contents

List of Figures

Chapter 1

Introduction

Honeypots is a valuable tool in the battle against cyberattacks. They act as lure for attackers and are able to capture crucial information for the nature and effects of attacks. The NoAH project focuses on building a flexible and scalable honeypot architecture. NoAH does not propose yet another centralized architecture but focuses on a modular design that enables distributed honeyfarms to cooperate and minimizes deployment overhead. The main advantage of NoAH is that offers the ability to people and organizations, who are not involved directly with the project, to participate. To achieve its goals, a set of mechanisms and tools have been designed (an extensive description can be found at [5]). In this document we describe the interface between all these components, their requirements along with the design of an operational database that will help in the management of NoAH's distributed infrastructure.

Giving a brief review of the NoAH architecture, we can identify the NoAH core and external components. The NoAH core is a distributed farm of honeypots, initially deployed by the NoAH consortium. Inside the NoAH core, we have both low- and high-interaction honeypots. Low-interaction honeypots serve as front-end to high-interaction ones and try to offload them from uninteresting traffic, like port scanning activities. High-interaction honeypots are instrumented machines that run virtual machines as a containment environment. We use Argos as our main containment environment. The components outside the NoAH core are honey@home and funneling/tunneling. Both of them aim at empowering people to participate to NoAH. Honey@home is a lightweight tool that listens an unused IP address and interacts with the NoAH core. All traffic directed to honey@home client is forwarded to the NoAH core, is processed by core honeypots and responses are injected back to the attacker. Funneling and tunneling are based on the same concept as honey@home and only have deployment differences. Their role is to help participating organizations, institutions and public bodies that want to share a larger unused IP address space. Traffic to that space is funneled to a single physical machine and then is tunneled to NoAH core through secure connections.

The interface between components must be well-defined for a smooth operation of both NoAH core and external components. In this document, we will define three interfaces : (a) low-interaction with high-interaction honeypots, (b) honey@home and funneling/tunneling with low-interaction honeypots and finally (c) high-interaction honeypots with signature generation components. Communication between low-interaction and high-interaction honeypots requires no modifications and is based on existing functionality. honey@home and funneling/tunneling interact with core honeypots through an additional component, called connection daemon. The connection daemon has double role. First, to accept incoming connections and unwrap packets that come through the tunnel. All connections are secured by using PKI cryptography (default SSL behavior). Second, to forward packets from clients to appropriate low-interaction honeypots and vice versa. Concerning interface between high-interaction honeypots and signature generation components, this is achieved through alert data produced by our main containment environment, Argos.

Additionally, we describe the design of NoAH database. The main role of this database is to keep track of honeypot configuration, data collected and information about participating organizations in the NoAH project. Information stored in the database allows easy deployment, resolves any conflicting configurations and provides an overview of NoAH core and external components. In the next Sections, after providing a quick overview of NoAH architecture, we will examine required interfaces one-by-one, study their requirements and describe the NoAH database design.

Chapter 2

Related work

In this Section we briefly discuss related work on both honeypot deployment architectures and available tools.

2.1 Deployment architectures

The Honeynet project [9] is a non-profit organization that is devoted to the research concerning honeypots and their underlying architecture. Central aim of the project is the in-depth analysis of attacks and the capture of malware (e.g. IRCBots). The Honeynet project deploys an architecture that consists of a central gateway, "Honeywall", and the honeypot network. Honeywall separates the network in which the honeypots are deployed from the rest of the network. Additionally, Honeywall performs access control of outbound connections from the honeypots and captures network data. The network behind Honeywall consists of high-interaction honeypots without emulation. Honeypots are also instrumented to trace their system calls through the Sebek tool [19].

HoneyStat project [25] aims at the detection and analysis of new attacks and zero-day worms and to apply statistical analysis to the worm behavior. The architecture is comprised of a central server (Analysis node) and distributed HoneyStat nodes. The HoneyStat nodes are the honeypot components that run multiple instances of emulated operating systems. If a compromise of an honeypot is detected, the captured data is sent to the Analysis node. HoneyStat nodes run VMware to emulate multiple instances of operating systems. Each operating system is assigned a set of 32 IP addresses.

The LEURRE.COM project [13] is an international project that operates a broad network of honeypots covering more than 20 countries. The primary aim of the project is to collect anonymous statistical data on a central server, gathered by low-interaction honeypots. Low-interaction honeypots run a modified version of honeyd [27], which emulated 3 operating systems (Windows 98, Windows NT and Redhat Linux). Periodically, all data gathered from the honeypot are sent to a central database server. The database can be afterwards queried about various

statistics. Except standard statistics, like the number of unique source IP addresses or distinct ports attacked, the database contains meta-data information. This meta-data information includes geolocation of IP address, passive OS fingerprinting of attackers and whether attackers scanned a honeypot sequentially or in parallel.

Collapsar [29] is a project developed by the Purdue University aiming at the deployment and management of a large number of coordinating high-interaction honeypots across different network domains. The Collapsar architecture is comprised of a *Collapsar center*, a centralized operation center which hosts a network of high-interaction honeypots, and *traffic redirectors*. The redirector allows to virtually deploy honeypots in arbitrary networks and its basic function is to forward all traffic received to the honeypots of the Collapsar center. The redirector is implemented as a virtual machine based on the User-Mode Linux (UML [21]). High-interaction honeypots of the Collapsar center are based on either VMware or UML.

In the Potemkin [28] honeyfarm infrastructure a large number of virtual high-interaction honeypots running on top of the Xen virtual machine is studied. Potemkin uses cloning and memory-page sharing techniques to run as many virtual honeypots as possible on a single machine, in order to avoid the need for numerous physical machines to emulate hundreds or possibly thousands of virtual honeypots. Virtual honeypots are recycled frequently (ideally once after a connection) as the longer the virtual machine runs the more resources it consumes. Thus, to run multiple virtual machines on the same physical host, recycling is needed. Outgoing traffic produced by honeypots is redirected to another honeypot of the honeyfarm to provide the attacker an illusion that she communicates with the real world.

Bailey et al. in [23] propose a hybrid honeypot architecture for scalable network monitoring. In their architecture they filter prevalent content by using low-interaction honeypots and use a handoff mechanism to enable interaction between low and high-interaction honeypots. Low-interaction honeypots filter out uninteresting traffic such as unestablished TCP connections or payloads that have been observed many times in the past. Apart from honeypots, their proposed architecture introduces a control component. This component aggregates traffic statistics from low-interaction honeypots and analyzes all received data for abnormal behavior.

2.2 Tools

Although this Section is not a complete list of available tools used for honeypot deployment, we present the most important ones that will help us define and implement the NoAH infrastructure.

The most well-known tool for setting up honeypots is honeyd [27]. Honeyd provides the functionality of a low-interaction honeypot as well as a framework for a virtual network of these honeypots. The attacker has the illusion that she communicates with hosts from an arbitrarily large network but in reality, she communicates with a single physical machine. Honeyd performs network stack emu-

lation for more realistic simulation of operating systems. It also simulates network services, such as popular web servers, and provides the support to act as a packet relay to external hosts. For those that want to avoid setting up a honeyd manually, HOACD is the implementation of a low-interaction honeypot, based on honeyd, that runs directly from a CD and stores its logs and configuration files on a hard disk. There are also tools that try to create emulation scripts for honeyd. Honeybee [7] is such a tool for semi-automatically creating emulators of network server applications. The resulting scripts can be used directly at honeyd. Similar effort is also presented in ScriptGen [26].

LaBrea Tarpit [12] is a low-interaction honeypot which tries to slow down the attacker. It answers connection attempts in such a way that the machine at the other end gets ”stuck”, sometimes for a very long time. KFSensor [11] is a commercial Windows honeypot, focusing on intrusion detection. It emulates Windows networking protocols and simulates vulnerable services and the existence of trojans. By acting as a decoy, it can aid firewalls and IDS systems by providing them information about possible intrusions. Another Windows-based solution is honeyBOT [8]. HoneyBOT is a medium-interaction honeypot and works by opening over 1000 UDP and TCP listening sockets which are designed to mimic vulnerable services.

The Google hack honeypot [6] (GHH) is another type of honeypot that aims at identifying search engine hackers. It is designed to provide reconnaissance against attackers that use search engines as a hacking tool. GHH permits search engines to index it and its virtual site contains multiple strings that are commonly searched by attackers.

Mwcollect [14] and nepenthes (soon they will be merged) are tools that collect malware from malicious URLs. They emulate vulnerable services and analyze the malicious payload to identify URLs. When a location is identified, they download and store locally the malicious software. Mwcollect also offers virtualized filesystems so that if during payload analysis a file needs to be written to disk, the real system is not affected. It also provides a virtual shell with limited capabilities like echo and batch processing.

VMWare [22], QEMU [17] and Bochs [1] are all virtual machine environments, on which any operating system can be set up. Virtual machines are used to deploy multiple virtual high-interaction honeypots on a single, or a few, physical machines. The virtualization allows to hide the underlying operating system which operates the virtual honeypots. As an advantage, the underlying operating system can be used to capture data, to detect compromises and to control the actions of an attacker on the virtual honeypot. Tools like Sebek [19] and Systrace [20] are used to monitor the activity of a honeypots, in terms of network traffic and system calls.

Chapter 3

Architecture overview

In the NoAH project we are not aiming to build yet another centralized infrastructure of honeypots. On the contrary, we would like to motivate all interesting parties to contribute to our knowledge against cyberattacks and help us implement working defense mechanisms. The distributed nature of NoAH makes its architecture challenging. In this Section, we will briefly discuss the NoAH architecture.

NoAH is structured around three main components: NoAH core, funneling/-tunneling and Honey@home. The overall architecture of NoAH can be seen at Figure 3.1.

NoAH core is the set of farms that include low- and high-interaction honeypots. As honeypots are the main detection mechanism, honeypot farms must be trusted entities and are maintained by NoAH partners. Low-interaction honeypots accept incoming connections to the core and filter out any unnecessary traffic. Interesting traffic is forwarded to high-interaction honeypots for inspection. The process of forwarding is called handoff. Handoff can be either assigned statically (e.g several honeypots will always run predefined services) or dynamically (new instances of Argos systems will be deployed dynamically based on pre-saved images of services). Static assignment is practical for a small number of services, while dynamic assignment scales better.

We use honeyd as the main software component of low-interaction honeypots. High-interaction honeypots are instrumented machines that usually run inside a containment environment, mostly virtual machines. We are using Argos, an instrumented version of Qemu that can efficiently detect zero-day attacks. The main idea behind Argos is to check if a process tries to execute data that came from the network.

Funneling and tunneling mechanisms allow organizations or institutes that are willing to cooperate with the NoAH project. Funneling is the process in which all packets destined to the black IP address space of the cooperating party are concentrated into a single physical machine. Tools like arpd help us to achieve funneling efficiently. Tunneling is responsible to forward these packets to the NoAH core. Packets are sent for inspection to one of NoAH's honeypot farms through a secure

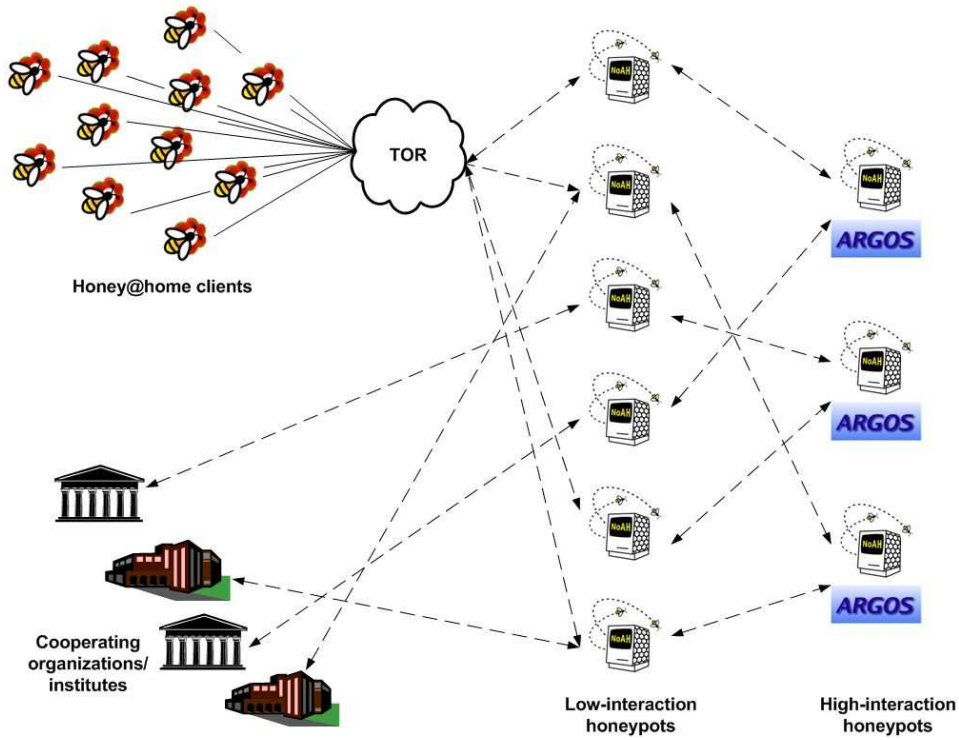


Figure 3.1: The NoAH architecture. Low-interaction honeypots serve as a front-end to cooperating organizations/institutes and honey@home clients while high-interaction honeypots perform attack detection

Figure 3.2: Honey@home running at the background

channel. Responses from honeypots follow the reverse path and are finally sent back to the attacker.

Honey@home is our main effort to motivate normal user to participate in NoAH. As ordinary users do not have the expertise around honeypot technologies, honey@home is designed as a simple tool that runs in the background. A screen-shot of honey@home can be seen at Figure 3.2. It is a very lightweight process that consumes no computational power and only a few megabytes of main memory (around 7 according to our measurements). Honey@home claims an unused IP address from the DHCP server and forwards all traffic destined to this IP to the NoAH core. As honey@home is a publicly available tool, it can be installed by anyone, including attackers. For this reason, we need to hide the identity of honeypots so attackers cannot directly attack them or overload them. Furthermore, we need to secure the identity of honey@home users so as attackers cannot use them for indirect attacks. Our choice of design was to use Tor, an anonymization system, that is already deployed and used by thousands of users. Tor offers both client and server anonymity.

Chapter 4

Low-interaction honeypots

4.1 Resource requirements

Low-interaction honeypots run lightweight software that requires minimal computational and memory requirements. From our experience with honeyd, a modern PC with 1 Gigabyte of main memory is sufficient. As per NoAH center we will not have more than 2 or 3 low-interaction honeypots, they can all run into the same machine. Furthermore, there is no need for extra storage requirements as low-interaction honeypots do not perform any logging operations. Concerning the operating system needed, any version of Linux can be used. We have successfully run honeyd on Redhat 9.0 and Debian. Necessary libraries to setup honeyd is Pcap and libnet.

4.2 Honeyd as a filtering component

Ideally, we would like high-interaction honeypots to process all traffic destined to black IP address space. However, as high-interaction honeypots are heavily instrumented machines, we need to offload them as much as possible. Low-interaction honeypots are used as front-end to high-interaction honeypots. Honeyd has the appropriate properties to play the role of the front-end and act as a filtering component. Filtering is defined as any action done to prevent high-interaction from being overloaded by packets that will not yield to any useful information. Honeyd can have a double role as filtering component.

First, it can absorb unestablished connections without overhead. A connection is considered unestablished when the SYN packet has been sent, honeyd has acknowledged it but afterwards connection initiator does not send any packet. This type of connections are typically seen in scanning activities, where scanning tools need only to verify that an IP address listens to a specific port and not fully communicate with the service. As honeyd performs stack emulation, unestablished connections are timed out after a certain period and finally dropped.

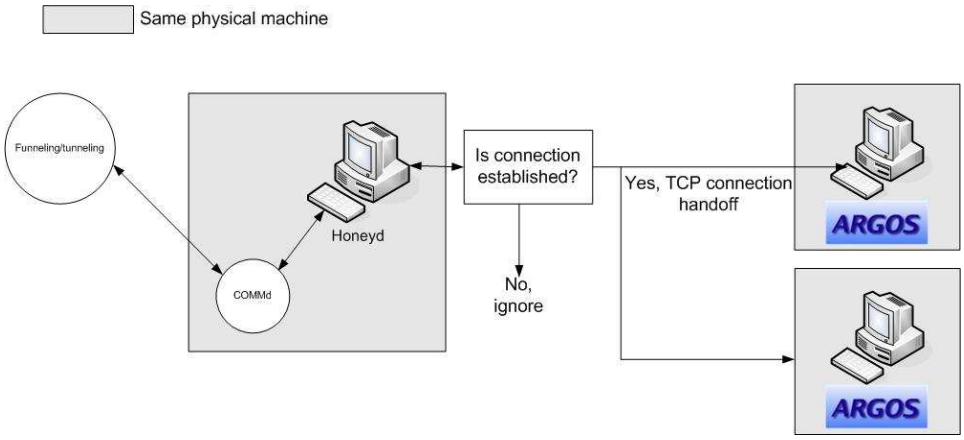


Figure 4.1: Handoff and filtering mechanisms inside the NoAH core

Second, honeyd can handle connections to specific ports, whose services can be accurately emulated. TCP port 5668 is an example. After monitoring a small portion of black IP address space for a month, we noticed high activity on that port, which corresponds to squid proxy. Attackers try to locate open proxies to access web anonymously or perform password guessing on web services. Other examples are SSH password guessing or telnet. Additionally, even non-accurate emulation scripts can be used in case high-interaction honeypots are overloaded, as they require minimal computational and memory resources.

4.3 Communication with Argos and other high-interaction honeypots

All connections that needs to be handled by high-interaction honeypots, and specifically Argos, will be forwarded by honeyd. The process of connection forwarding is called hand-off. We have implemented two types of hand-off inside honeyd. The first one is based on the destination port, e.g packets that are destined to port 80 will be handed-off to Argos system A while packets to port 139 to Argos system B. The second one is based on patterns found inside packet payloads along with destination port (port is optional). An example is that packets which contain the pattern ”/admin.html” will be forwarded to an Argos system that runs a Web server. This advanced functionality is needed mainly to either redirect or drop known attack vectors to specific honeypots.

Hand-off is performed upon the connection with the attacker is established. The process, along with filtering of unestablished connections, is illustrated in Figure 4.1. Honeyd opens a socket with Argos and sends all application data received by the attacker through this socket. As application data, we define the TCP (or UDP) payload. For the time being,we have chosen to have a persistent socket between honeyd and Argos to avoid the overhead of creating a new connection for

every incoming attack. Note that as the connection between honeyd and Argos is a normal TCP connection, there is no need for an extra communication component on the Argos side or any other high-interaction honeypot we may deploy.

The main problem of hand-off is that the low-interaction honeypot and the Argos system have different IP addresses. In cases where application protocol embeds information about IP addresses, like FTP passive, this mismatch on IP addresses will cause the application not to work properly. To overcome this problem, honeyd replaces all occurrences of its IP address inside packets (either in human readable or binary format) with the IP address of Argos system. Note that this change cannot be applied generally in any protocol but works well for the protocols we have tested; FTP, HTTP and SSH. Also encrypted protocols that require IP address information, like secure FTP, cannot be properly handed-off.

Chapter 5

Interface between Honey@home, funneling/tunneling and NoAH core

The NoAH infrastructure is not a set of central honeypots running dedicated services. We have proposed to extend the monitored black IP address space by engaging all interested parties to the NoAH project. To achieve this, Honey@home and funneling/tunneling components have been developed. Honey@home is a simple software tool that can be installed in any home computer and monitor an unused IP address.All traffic captured by honey@home is then processed at the NoAH core. Funneling/tunneling is a technique, focusing mainly on participating organizations that want to monitor larger black IP address space. Funneling gathers all packets destined to unused IP addresses to the same physical machine while tunneling sends these packets to the NoAH core through a secure channel. This section aims at describing the communication components between Honey@home, funneling/-tunneling and the NoAH core.

5.1 Communication components

Honey@home client runs at the background and monitors an unused IP address. It is available for both Linux and Windows operating system. For the Linux version, pcap and openssh libraries are needed. For the Windows version, all missing libraries will be installed automatically upon software installation. To preserve the anonymity of both honey@home clients and the honeypots of NoAH core, an anonymity system is used. The anonymity system of our choice is TOR, as it is already deployed and preserves good properties, like low latency, both client and server anonymity and it is robust against attack on its infrastructure. As TOR is subjective to timing attacks, that is correlating traffic between entry and exit point of an anonymous path, we have suggested running our own TOR server on the server side. For more details, you can refer to Deliverable 1.1. The TOR server

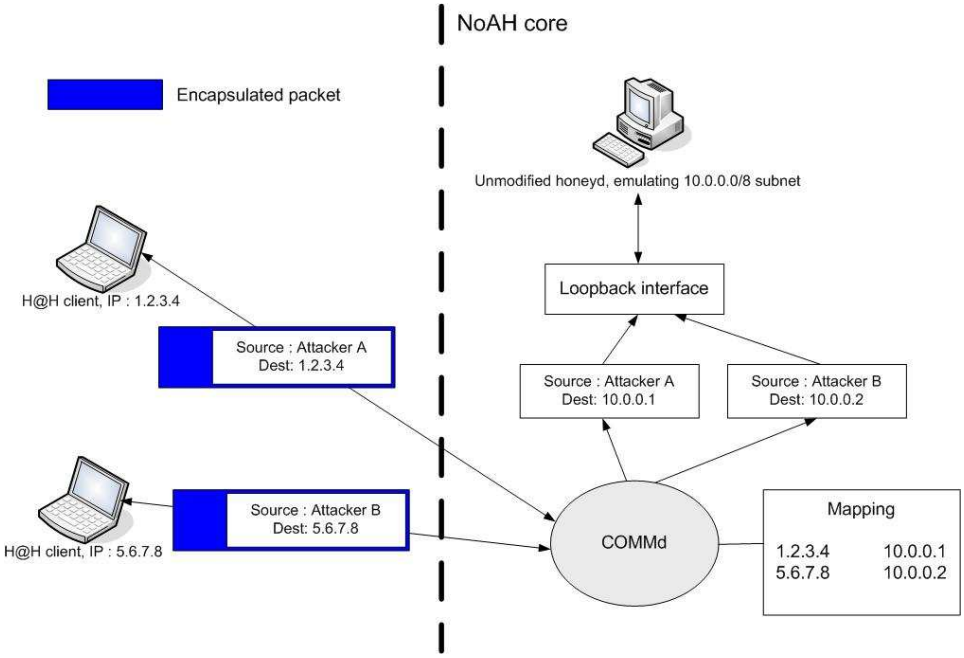


Figure 5.1: The communication path between honey@home clients and the NoAH core

runs on a dedicated machine with Linux operating system installed. On server side, apart from TOR server, a connection daemon is deployed to handle communication with honey@home clients. This daemon is lightweight and can run in the same physical machine with low-interaction honeypots. The connection daemon is available only for Linux operating system. The connection daemon has double role. First, to accept incoming connections and unwrap packets that come through the tunnel. All connections are secured by using PKI cryptography (default SSL behavior). Second, to forward packets from clients to appropriate low-interaction honeypots and vice versa. As low-interaction honeypots are located in the same physical machine, daemon injects packets to loopback interface and reads packets from the same interface. In order to support clients that are located behind NAT, and some of them will probably have the same –internal– IP address, daemon is also responsible to rewrite packets and keep state for the public-translated pair.

A view of the communication components is displayed at Figure 5.1. When honey@home client starts, it establishes a connection with connection daemon through TOR. Later on, The attacker sends a packet to the unused IP address A. Honey@home captures the malicious packet and sends it to NoAH core. The connection daemon keeps the mapping that client with public address B and internal address A (in non-NAT cases A and B will be the same) is mapped to address 10.0.0.5. It then rewrites the source address of packet and injects it to the local interface where honeyd listens. Responses from honeyd have the destination ad-

dresses rewritten and are forwarded by the connection daemon back to the client, which in his turn sends the response back to the attacker. This implementation is scalable as connection daemon can send the packets to any honeypot (either at the same physical machine or at the same subnet).

Funneling/tunneling component is based on a modified version of Honey@home client, called Honey@home Pro. The differences between normal honey@home client and Pro version are two. The first one is that Pro client does not take an IP address statically or through DHCP but it is based on a BPF filter. Thus, the unused address space must reach the Pro client through an external mechanism, like arpd. Arpd is a utility that responds to ARP requests and claims unused IP addresses. The second one is that packets reaching Pro client are not sent to the NoAH core through TOR but directly through a SSL channel. On the client side, only honey@home Pro and arpd are needed. They are both available for Linux and Windows operating systems. On the server side, the communication components are the same with those of honey@home, that is the connection daemon.

5.2 Logging format and facilities

Honey@home clients are instrumented to keep a tcpdump trace of all traffic destined to their black IP address space. This logging is done mainly for debugging reasons but these traces can also be used for more detailed examination of attacks and replay at a later time. As COMMd is expected to receive aggregate traffic from all clients, its logging component is disabled by default. We have also implemented logging during handoff performed by honeyd but it is also disabled by default. We have chosen tcpdump format to store our traces for two reasons. The first one is that is a common format and there are plenty of tools to read and extract statistics from tcpdump traces. Their standard format makes them also suitable for public access after being anonymized. The second one is that we can easily replay them at any given time using existing tools. Replay can help us calibrate our detection methods at any desirable speed.

Chapter 6

Using *Argos* as a high-interaction honeypot

In this section we will briefly discuss how *Argos* can be used as a high interaction honeypot. First of all, we will list resource requirements and present performance evaluation results. Next, we will mention few example operating system configurations supported by *Argos.* Finally we will explain format of logs produced by *Argos* containing footprints of the attack.

6.1 Resource requirements

In this section we will briefly discuss the requirements one has to fulfill in order to setup *Argos*, the NoAH high-interaction honeypot.

Argos source code (argos.0.1.5.tar.gz) can be downloaded from the official *Argos* website: `https://gforge.cs.vu.nl/projects/argos/`.

To setup *Argos* so that it runs on the host operating system, and shares the same network environment as the host, e.g., the standard DHCP server using the bridge interface, one needs a host operating system with the following features:

- a recent Linux kernel with support for Network bridge and TUN device drivers,
- bridge-utils,
- SDL library [18].

To setup a guest operating system to use with *Argos*, one needs:

- a CD/DVD or a CD/DVD ISO image including a guest OS,
- up to 3 GB of free disk space depending on the operating system one is trying to install.

6.2 Performance

We implemented an *Argos* prototype on top of the Qemu [24] open source emulator. In this section we will briefly discuss some preliminary results with the prototype. A more thorough evaluation will be presented in the upcoming deliverables.

Performance evaluation was carried out by comparing the observed slowdown at guests running on top of various configurations of Argos and unmodified *Qemu*, with the original host. The host used during these experiments was an AMD Athlon™ XP 2800 at 2 GHz with 512 KB of L2 cache, 1 GB of RAM and 2 IDE UDMA-5 hard disks, running Gentoo Linux with kernel 2.6.12.5. The guest OS ran SlackWare Linux 10.1 with kernel 2.4.29, on top of *Qemu* 0.7.2 and Argos. To ameliorate the guest's disk I/O performance, we did not use a file as a hard disk image, but instead dedicated one of the hard disks.

To quantify the observed slowdown we used bunzip2 and Apache. bunzip2 is a very CPU intensive UNIX decompression utility. We used it to decompress the Linux kernel v2.6.13 source code (approx. 38 MB) and measured its execution time using another UNIX utility `time`. Apache, on the other hand, is a popular web server that we chose because it enables us to test the performance of a network service. We measured its throughput in terms of maximum processed requests per second using the httperf HTTP performance tool. httperf is able to generate high rates of single file requests to determine a web server's maximum capacity.

In addition to the above, we used BYTE magazine's UNIX benchmark. This benchmark, nbench for brevity, executes various CPU intensive tests to produce three indexes. Each index corresponds to the CPU's integer, float and memory operations and represents how it compares with an AMD K6™ at 233 MHz.

Figure 6.1 shows the results of the evaluation. We tested the benchmark applications at the host, at guests running over the original *Qemu*, and at different configurations of *Argos*: using a bitmap, and using a byte-map with code-injection detection enabled. These are indicated in the figure as Vanilla QEMU, Argos-B, and ARGOS-B-CI respectively. The y-axis represents how many times slower a test was, compared with the same test without emulation. The x-axis shows the 2 applications tested along with the 3 indexes reported by nbench. Each color in the graph is a configuration tested, which from top to bottom are: unmodified *Qemu*, *Argos* using a byte-map for memory tagging, and the same with code-injection detection enabled. Apache throughput in requests served per second is also displayed in Table 6.1.

Even in the fastest configuration, *Argos* is at least 16 times slower than the host. Most of the overhead, however, is incurred by *Qemu* itself. *Argos* with all the additional instrumentation is at most 2 times slower than vanilla *Qemu.* In the case of Apache and float operations specifically, there is only an 18% overhead. This is explained by the lack of a real network interface, and a hardware FPU in the emulator, which incurs most of the overhead. In addition, we emphasize that we have not used any of the optimization modules available for *Qemu.* These modules

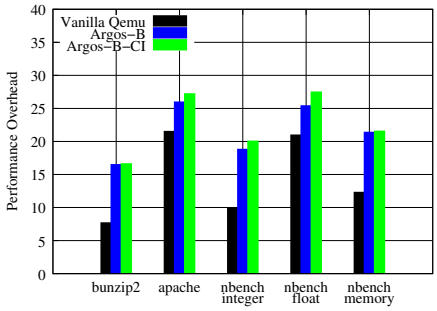


Figure 6.1: Performance benchmarks.

Configuration	Served Requests/Sec.
Native	499.9
Vanilla Qemu	23.3
Argos-B	18.7
Argos-B-CI	18.3

Table 6.1: Apache throughput.

speed up the emulator to a performance of roughly half that of the native system. While it is likely that we will not quite achieve an equally large speed-up, we are confident that much optimization is possible.

6.3 List of emulated services

In principle since *Argos* is implemented on top of *Qemu* [24], it supports exactly the same operating systems as the vanilla version of the emulator. Nevertheless, to aim signature generation process we employ a novel technique to automate forensics on the code under attack. Currently, forensics are available for both Linux and Win32 systems. We emphasize however that even without the shellcode, which by its nature may contain OS-specific features, Argos still works, albeit with reduced accuracy.

Qemu emulates multiple architectures such as x86, x86_64, POWER_PC64, etc. Our implementation extends *Qemu*'s Intel architecture.

Table 6.2 lists few example configurations on which *Qemu* was tested and is fully working[1].

[1]The table is based on `http://www.claunia.com/qemu/`

Guest Side			Host Side	Full *Argos* support[2]
OS	Architecture	Version		
Knoppix	PC (IA-32)	4.0.2	Linux	yes
NetBSD	PC (IA-32)	2.1	Linux	no
OpenBSD	PC (IA-32)	3.8	Linux	no
Ubuntu	PC (IA-32)	5.10	Linux	yes
Ubuntu Server	PC (IA-32)	5.10	Linux	yes
SlackWare Linux	PC (IA-32)	10.1	Linux	yes
Windows XP	PC (IA-32)		Linux	yes
Windows 2K	PC (IA-32)		Linux	yes

Table 6.2: Example configurations supported by *Argos.*

6.4 Logging requirements/format

In this section we will explain what useful information is extracted by *Argos* once an attack is detected.

The logs produced by *Argos* contain the memory fingerprint of a detected attack. All tainted system state is exported. Argos isolates the attacked process by only looking up memory pages accessible to it. This way the volume of exported data is reduced to exclude data that cannot be associated with the attack. Furthermore, when provided with a hint about the guest operating system (*Argos* can be run with -Linux, -win2k, or -winxp options), it is able to distinguish between user and kernel memory space further reducing the volume of exported data.

Logs are written to a file in the working directory following the naming format shown below (`rid` is a randomly generated integer ID identifying the detected attack): argos.csi.[rid].

To aid signature generation, *Argos* can be configured to track incoming network data. This means that *Argos* can track exactly to which byte in the network trace each tainted byte corresponds from the moment it enters the system via the network card. Depending on the *Argos* version configured (*standard* vs. with *network tracking* enabled) log formats might differ. In the rest of this section each version will be explained in detail.

All *Argos* logs follow a strict format:

- it begins with a log header,
- the log header is followed by descriptions of multiple memory blocks, each containing the following parts:
  - a memory block header,
  - contents of the memory block,
  - network indices of the successive bytes within the memory block being described (only if *Argos* was configured with network tracking enabled, and the memory block is tainted)

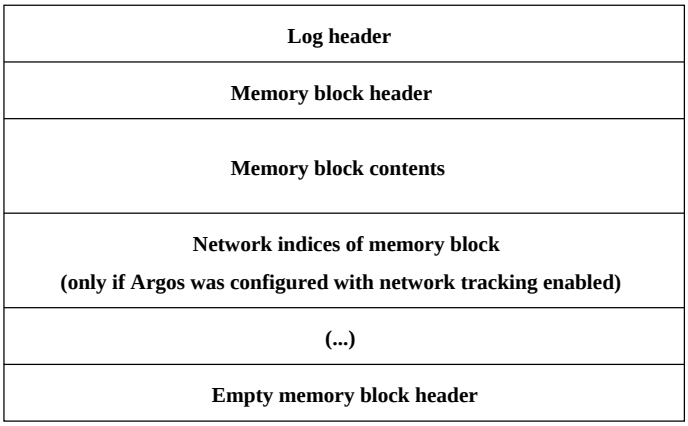


Figure 6.2: Log high-level format.

Format	Arch	Type	Timestamp
Register contents			
Register memory origins			
Register network indices			
EIP	EIP tag		EFLAGS

Figure 6.3: Log header format. (Fields containing network indices are valid only if *Argos* was configured with network tracking enabled.)

- the end of the log is marked by an empty memory block header.

The structure of a dumped data is shown in Figure 6.2.

6.4.1 Log header

The structure of log header is presented in Figure 6.3. All 2, 4, and 8 bytes fields are in little endian format.

Format. The header always starts with the *Argos* version number that determines the format of the rest of the log header.

Architecture. The second field specifies the emulated architecture that produced the log. It can be equal to either ARGOS_ARCH_I386 (value 0) or ARGOS_ARCH_X86_64 (value 1). The architecture field also affects the rest of the data in the header and log.

Name	Value	Description
ARGOS_ALERT_JMP	0	A tainted value was used in an indirect jmp instruction.
ARGOS_ALERT_LJMP	1	A tainted value was used in a protected mode jmp instruction. (Obsolete. It will be merged with `jmp` instruction.)
ARGOS_ALERT_TSS	2	A tainted value was used to load EIP within TSS (Task State Segment).
ARGOS_ALERT_LCALL	3	A tainted value was used in a real mode call instruction.
ARGOS_ALERT_IRET	4	A tainted value was used in an iret instruction.
ARGOS_ALERT_RET	5	A tainted value was used in a ret instruction.
ARGOS_ALERT_WRMSR	6	A tainted value was written in a MSR register. (Obsolete. It will be completely removed.)
ARGOS_ALERT_CI	7	Execution flow was redirected to a tainted instruction.

Table 6.3: *Argos* alert types.

Type and time-stamp. The following fields specify the type of the alert that generated the log, and the time it was issued. Table 6.3 shows all possible alert types.

Register contents. The rest of the header's fields are dependent upon the architecture. First we can expect the contents of the general purpose registers. x86 architectures have 8 32 bit registers, while x86_64 architectures have 16 64 bit registers. The sequence the registers are written out is the following: EAX, ECX, EDX, EBX, ESP, EBP, ESI and last EDI.

Register memory origins and network indices. Register contents are followed by corresponding memory origins and possibly network indices (both being fields of register tags). Depending on the *Argos* version configured register tags have the following format: (target_ulong is an unsigned long on the emulated architecture)

- *Argos* with network tracking *disabled*: the tag is used to store the physical memory address from where the contents of the register originate: struct argos_rtag target_ulong origin; /* memory origin */ ;
- *Argos* with network tracking *enabled*: the tag is used to store both the physical memory address and the network index from where the contents of the

register originate: (the network index is calculated as the offset from the beginning of the network log) struct argos_rtag target_ulong origin; /* memory origin */ uint32_t netidx; /* network index */ ;

We can see that each tag indicates the following features: (1) whether the register contents are tainted, (2) where this value was loaded from in physical memory (if it is tainted), and (3) where this value was loaded from in the network log (if it is tainted, and if *Argos* was configured with network tracking enabled). There is one tag for each general purpose register and they have the same size. If the value of the tag is non-zero, then the contents of the corresponding register are tainted, and corresponds to the physical memory address where the tainted data were read from and also indicate index of the tainted data in the network log. If the register contents have been directly read by the network, the value of the memory origin is -1 (0xffffffff).

EIP. After the general purpose registers we find the instruction pointer EIP along with its tag. Normally, the tag for EIP should be tainted for an alert to be issued, and EIP should contain a memory address provided by the attacker (usually the address where shellcode was injected).

EFLAGS. Finally, the last field of the header is the value of the EFLAGS register (size also depends on architecture).

Summary. Table 6.4 summarizes the header's fields. For reference, the C structure used within *Argos* is shown below (CPU_NB_REGS is the architecture's number of general purpose registers, and ARGOS_NET_TRACKER is defined if *Argos* was configured with network tracking enabled):

```
struct argos_log_hdr_struct {
    uint8_t format;
    uint8_t arch;
    uint16_t type;
    uint32_t ts;
    target_ulong reg[CPU_NB_REGS];
    target_ulong rorigin[CPU_NREGS];
#ifdef ARGOS_NET_TRACKER
    uint32_t rnetidx[CPU_NREGS];
#endif
    target_ulong eip;
    target_ulong eiporigin;
#ifdef ARGOS_NET_TRACKER
    uint32_t eipnetidx;
#endif
    target_ulong eflags;
```

Field name	Length (Bytes)	Description
FORMAT	1	Format version.
ARCH	1	Architecture (I386 or X86_64).
TYPE	2	Type of attack. Its value is one of the *Alert types* in Table 6.3.
TIMESTAMP	4	Timestamp of the attack.
REGISTER VALUES	32(I386) or 128(X86_64)	Values of the 8(I386) or 16(X86_64) registers.
REGISTER ORIGINS	32(I386) or 128(X86_64)	Memory origins of the registers.
REGISTER NETWORK INDICES	32 (I386) or 64 (X86_64)	Network origins of the registers. (Only if networking tracking is enabled in *Argos*).
EIP	4 (I386) or 8 (X86_64)	Value of the instruction pointer.
EIP ORIGIN	4 (I386) or 8 (X86_64)	Memory origin of the instruction pointer.

Table 6.4: Log header fields specification.

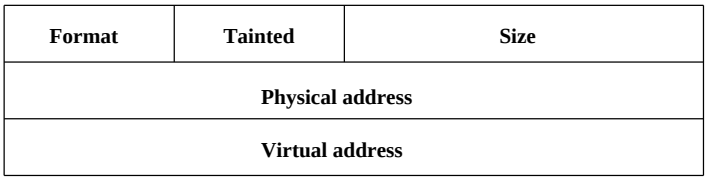


Figure 6.4: Memory block header format.

```
} __attribute__((packed));
```

6.4.2 Memory block header

The structure of the memory block header is presented in Figure 6.4.

Format. The header always starts with the *Argos* version number that determines the format of the rest of the header.

Tainted. The second field is a flag that specifies whether the memory block is tainted or not. For most of the blocks this flag is asserted, there is the case though that EIP points to a memory area that is not tainted. In that case the memory block is written to the log and the taint-ness flag is set to zero.

Field name	Length (Bytes)	Description
FORMAT	1	Format version.
TAINTED	1	Taintness flag.
TYPE	2	Size of block's contents.
PHYSICAL ADDRESS	4 (I386) or 8 (X86_64)	Physical memory address of block.
VIRTUAL ADDRESS	4 (I386) or 8 (X86_64)	Virtual memory address of block.

Table 6.5: Log header fields specification.

Size. The third field specifies the size of the block's contents that follows the header.

Physical and virtual addresses. The memory address of the block is next. Both the physical and virtual address of the block are written out. If a certain memory block is mapped to multiple virtual memory addresses then the block will be written out multiple times.

Summary. Table 6.5 summarizes the memory block header's fields. For reference, the C structure used within *Argos* is shown below:

```
struct argos_mblock_hdr_struct {
    uint8_t format;
    uint8_t tainted;
    uint16_t size;
    target_ulong paddr;
    target_ulong vaddr;
} __attribute__((packed));
```

Chapter 7

Signature generation components

The signature generation component consists mainly of the *connection tracker* software. The *connection tracker* is a modular component used to capture and process network traffic in order to extract meaningful information for the generation of signatures. In the context of NoAH, the component should be installed either on the high interaction honeypots or on separate machines attached to their network uplinks. A very basic view on the functionality of the *connection tracker* is displayed in the following figure:

From this follows that its functionality is basically that of a protocol analyzer. But a protocol handler is not only responsible for keeping and logging state. It is the custom information that makes the difference. Each handler has to specify the type, length(distribution) and byte frequency distribution of all of the fields of the handled protocol.

In order to generate signatures, the information provided by this component has to be combined with information provided by Argos. As can be extracted from the section about the Argos component, if network tracking is enabled in Argos, it is able to pinpoint some bytes in the received network traffic that triggered its detection mechanism. By using these bytes and eventually some bytes around them as descriptive pattern, we already dispose of the simplest type of a signature: a pattern based signature. With such a signature, we only have to look for such patterns in network traffic to identify attacks. Unfortunately, this may cause a large amount of false positives because a specific pattern is likely to occur in legitimate traffic too. An efficient way to improve this kind of signature is to reduce the amount of traffic against which the pattern is matched. This is easy if we know for example in which protocol field the offending pattern was found. We would then have to search for this pattern only if incoming network traffic belongs to this field. Additionally, if we have as provided by the *connection tracker* component, type, length(distribution) and byte-frequency distribution information of the corresponding field, we can narrow down the amount of traffic that we have to check for a certain pattern significantly. It would also be possible to create signatures that do not depend on a certain pattern but on byte-frequency distributions or length

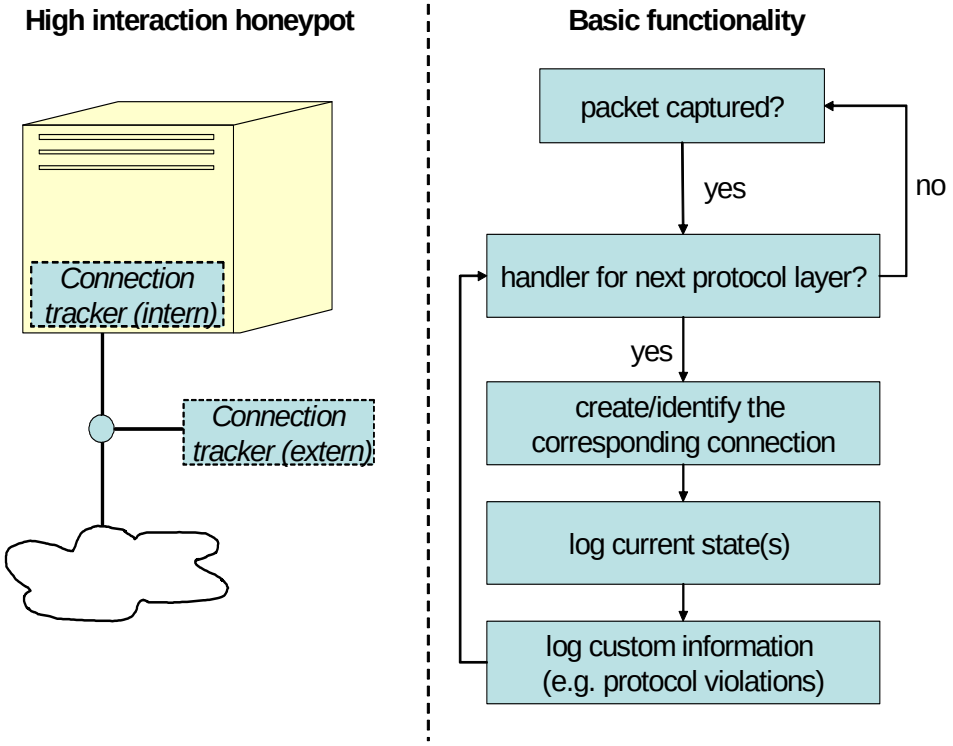


Figure 7.1: Overview of connection tracker

information. But why implement another protocol analyzing component? There exist already many! The reason is that to our knowledge there is no software that suits our needs. Therefore we decided to implement our own analyzer. Since it is for proof-of concept only, only a limited amount of handlers will be implemented. But the flexible structure allows for an easy integration of additional handlers.

7.1 Interface with signature generator and Argos

From the previous section follows that we need to be able track down the packet containing the offending bytes in the network traffic as much as their offset in the packet. An additional information that we need is the timestamp of the detected attack. With the packet and the timestamp we are able to link it to the information stored by the *connection tracker* component. In case the *connection tracker* component is installed on the same machine as Argos, an ad-hoc way to get this information from Argos is the following:

1. Set a watchdog for Argos. If Argos terminates, continue with step 2.
2. Check if there is a (new) entry to the log written by Argos (file: argos.csi.[rid]).
3. Get TIMESTAMP field and REGISTER NETWORK INDICES (while taking the ARCH field into account) from the log.
4. Parse the network traffic dumped by Argos to get the REGISTER NETWORK INDICES corresponding packet and bytes.

A less ad-hoc solution would require Argos to send an alert to the *connection tracker* component whenever an attack is detected using socket communication. This alert would ideally include already the corresponding network packet, timestamp and involved bytes. If the *connection tracker* component is not located on the same machine as Argos, this is a mandatory feature. Its implementation including the specification of the alert format would be part of work package 2.2.

7.2 Signature export

It is not yet decided if and which commonly used signature types the *connection tracker* component will support. But if we consider todays more or less widely used freely available intrusion detection/prevention systems, we identify the following two signature types: Snort®and Bro. A lot of other signature types have been proposed like e.g. in the Shield project from Microsoft®Research but most of them are only implemented in some proof-of-concept prototypes which are not widely used if available at all. On the one hand, the reason for this diversity is certainly the limitations of Snort and Bro signatures and on the other hand the lack of a ready to use implementation of one of the more powerful approaches. Since our goal is to automate signature generation and to generate signatures that could

be used with common intrusion detection/prevention systems, the development of yet another signature type is not in our focus. In fact, in a first step we combine the information provided by Argos with the information from the *connection tracker* component. This information pool forms the basis for the generation of e.g. Snort, Bro or custom signatures. In a second step we plan to use it to create Snort and/or Bro signatures.

Chapter 8

NoAH Database Architecture

The integration of the different NoAH components into a homogeneous architecture is crucial for the deployment of the project. However, beside the technical integration of the NoAH components, solutions to organizational issues have to be addressed to be able to operate a fully functional infrastructure. These issues include the following aspects:

- Since the NoAH network of sensors will be distributed among different, independent sites, contact information for these sites are required to timely inform them about technical or organizational problems.

- The NoAH architecture is designed to flexibly react to attacks by dynamically assigning the appropriate advanced sensors (e.g. Argos nodes) to the low-interaction components (e.g. honeyd nodes). This can only be done if the current status information and configuration of each sensor is known.

- The NoAH sensors of the demonstrator will output a large amount of data related to seen attacks. This data will include the attacks itself and data resulting from the analysis of these attacks (e.g. signatures). Technical and organizational solutions are required for an efficient storage and secure access to the resulting data. The technical and organizational solution should also address anonymity or privacy issues arising from security policies as well as national laws of the participating sites.

It points out that all these issues either affects the administration or the operation of the demonstrator. Therefore, we propose that two roles are required for the operation of the demonstrator dedicated to the administration and operation. In addition, a user role is needed taking care of the analysis and presentation of the resulting data.

In the following section we will describe the objectives of these roles in detail and use this basis to derive technical requirements ensuring a smooth operation of the demonstrator. To anticipate the results, it points out that a relational database is

perfectly suited to met these requirements. The database design is described in the section 8.2.

8.1 Roles for the NoAH demonstrator

Basically, the roles can be divided into the administrative, operational, and user role.

The objectives of the user role are:

- The value of the NoAH demonstrator for the security community will depend on the quality of the results. In other words, the demonstrator is nothing without the presentation of accurate and reliable results. The analysis and the presentation of the results is the objective of the user role.
- As an important requirement the NoAH demonstrator has to meet some restrictions concerning the storage and access of the resulting data. These are:
  - It has to be in accordance with national laws concerning the usage of personal data.
  - The demonstrator has to meet the security policies of the NoAH project as well as the participating sites with respect to the storage and access of data.

Both requirements need organizational as well as technical solutions. To address the organizational we propose to introduce a ”Data Usage Agreement (DUA)” and a ”Non-disclosure Agreement (NDA). The Data Usage Agreement addresses all issues coming from the needs of the NoAH partners and cooperating sites. These may include that all data which is supplied by a site belongs to this site. As a consequence all data belonging to a site has to be erased if requested by this site. An additional common agreement in honeynet projects is that a site can only access confidential data in anonymized form except of their own data. In addition, it has to be specified in which form data can be used in publications. Beside the data supplied by the sensors, there is other confidential data including for example the names of the cooperating sites or the locations of the sensors. To protect this data is objective of the NDA. For example, some sites may prefer to hide their identity when cooperating with the NoAH project. Thus, this knowledge has to be kept confidential by the NoAH partners. A first proposal for both agreements can be found in the draft policy included in deliverable D0.2 [4].

It should be noted that these requirements have some implications on the technical solutions of the NoAH demonstrator. For example, if a site can request to erase all data belonging to them the data has to be tagged in some form at the import.

The administrative role covers the following tasks:

- An important task of the administrative role is to integrate new sensors into the NoAH demonstrator. This includes the technical integration of the sensor as well as the organizational issues as the acceptance of the code of conduct and contact information or the agreement about anonymization issues.

- Access to the data must be limited in agreement with the policy according to the owner of the data and the accessing site or user. As a consequence, the accessing user or site has to be authenticated and the owner of the data must be known. Objective of the administrative role is to maintain the credentials of the users and granting new users access to the data (e.g. by issuing S/MIME certificates).

- There are some reasons why it may be necessary to get in contact with these sites. First, these sites have to be informed if a sensor needs attention because of hardware or software problems that require physical access to the machine.

- An important objective of the administrative role is to maintain information about the configuration and status of all low- as well as high-interaction honeypots. The NoAH architecture is designed to use low-interaction honeypots to accept all incoming connections. For the analysis of the corresponding attacks these connections are dynamically relayed to high-interaction components. As a consequence, the demonstrator has to decide in real time to which honeypot the connection is relayed. This decision can only be taken if the configuration and status of all available honeypots is known. For example, if a low-interaction honeypot accepted a connection on a typical windows port (e.g. TCP/135) it is pointless to pass the connection to a high-interaction honeypot which do not support this service. In addition, it is necessary to know which high-honeypots are in a clean state and can accept data.

The objectives of the operational role are

- The operational role will enable the users to access the data resulting from the demonstrator according to the access policy. As previously mentioned assertion of the access policy requires the user to be authenticated and the site must be known owning the data. The provision of the interfaces for data access and the assertion of the access policy is objective of the operational role.

- As previously mentioned, the demonstrator has to dynamically decide which high-interaction honeypot is assigned to a low-interaction honeypot receiving unknown network data. This is the objective of the operational role. This role also maintains log-data concerning these assignments. These information enables afterwards to reconstruct the source from which the high-interaction honeypot was attacked.

- To be able to dynamically assign a high-interaction honeypot the configuration as well as the status of each honeypot must be exactly known. Note, that the maintenance of the configuration information is part of the administrative role. In contrast to the configuration, the status of a honeypot changes dynamically with the operation of the demonstrator. Therefore, maintenance of these state information is assigned to the operational role. We propose to use the following states for each honeypot:
  Idle. The honeypot is clean and running and can assigned to a NoAH low-interaction component.
  Down. The honeypot is not running.
  Maintenance. The honeypot has been attacked and must be re-installed in order to change its state to ”idle”.
  Compromised. Non-legitimate actions originating from a honeypot have been observed. Therefore, it is susceptible of being compromised in a non-foreseen way.
  Susceptible of denial of service. If a sensor is permanently under a very heavy load without censoring any exploit data a denial of service attack can be assumed and the honeypot will be set to this state.

  To keep track of the state of each honeypot is objective of the operational role.

- An additional objective is to detect and react to attacks and technical problems. The NoAH demonstrator can be assumed to be an attractive target for attackers, especially intended to prevent the demonstrator from being operative (denial of service). Most likely the attacker would try to exhaust a specific resource of the demonstrator. This can for example be done by sending a very massive number of packets to one or multiple low-interaction honeypots in order to pollute the subsequent high-interaction honeypots. To be able to detect this kind of attack is very important for the survivability of the demonstrator. If such an attack is assumed the low-interaction honeypots which are susceptible of being the source or relay of this attack can be deactivated.

  However, not all problems are caused by attacks. Hardware or software failures can end up in the same or similar effects. Therefore, the NoAH architecture should foresee a component monitoring the state of the demonstrator and reacting to problems. This component will also be important to optimize the interaction between the different components of the NoAH architecture.

- One of the major aims of the project is to detect attacks and to generate signatures characterizing these attack. An objective of the operational role is to maintain a database including all generated signatures and the corresponding attacks. As an important advantage, this database enables to decide whether an attack has already been been seen or not. Of course, attacks targeting

to abuse an unknown vulnerability are of much more interest compared to already known attacks.

To provide the results to the NoAH partners and cooperating teams an interface to this database have to be established that allows secure access to the data.

Beside the signature itself this database also cover information about the honeypot that was attacked and its configuration. This allows to reconstruct the vulnerable software versions.

8.2 Database design

The roles incorporated in the deployment of the demonstrator have been described in the previous section. All roles rely on a fast, flexible, and reliable access to data concerning the sensors and the resulting data. As a consequence, the demonstrator is heavily dependent on the configuration and status data of the honeypots. For example, a realistic requirement will be to find out in milliseconds which high-interaction honeypots are available running a Windows XP Service Pack 2 operating system with an IIS 5.0 Server that are in an ”idle” state. In addition, the demonstrator will produce a large amount of resulting data. However, this data is nearly worthless without a fast and flexible way to analyze and access it. It points out, that these requirements are perfectly met by an relational database. In this section we propose a database design well suited to support the previously mentioned objective and roles.

The database architecture is comprised of the following components:

- Contact database information
- Honeypot configuration and status information
- Honeypot assignment information
- Netflow data
- Alert data
- Signatures and resulting data from honeypots

Each component is comprised of a set of information entities as described below and will consist of a set tables when the database architecture is finally specified:

8.2.1 Contact information

Objective of the contact database is to maintain contact information of the cooperating sites. This data includes the following information for each site:

- Identity of site
- Administrative and technical Contact (e.g. email and postal address).
- Identities of sensors run by this site.
- Anonymization policy concerning data from this site (e.g. no IP source address).

8.2.2 Honeypot configuration and status information

This component is designed to maintain the following information for each honeypot:

- Identity of honeypot or sensor
- Type of honeypot or sensor
- IP Address or address range
- Export restrictions due to anonymization policy
- Configuration of honeypot
- Current state

Type of the honeypot include for example honeyd or nepenthes low-interaction honeypots or Argos node.

We propose to specify the software environment by a simplified Common Model of System Information (CMSI) as described in [3]. This model is designed to specify the version of the operating system and service or program in a structured form. Although it is intended for the specification of the affected system and software in form of a XML document, its structure can be transformed into a relational database scheme. This format also allows to specify the simulated operating system (honeyd) and the simulated services (honeyd and nepenthes).

For the list of the values of the honeypot states we refer to section 8.1.

8.2.3 Honeypot assignment information

As mentioned in section 3, high-interaction honeypots are dynamically assigned to analyze connections established to low-interaction honeypots. Maintaining these information is the objective of this database component. Its scheme covers the low-interaction honeypot (or sensor) identification number and the high-interaction honeypot identification number.

In addition these assignments including time-stamps should be recorded in a log-file (e.g. syslog facility) to be able to reconstruct all previous assignments.

8.2.4 Netflow data

This data is primarily intended to be used either to detect attacks against the NoAH infrastructure itself or to find performance shortcomings of the demonstrator. Based on the experiences of other projects like the Carmentis [2] project it seems to be advantageous to store the data in a round robin database as underlying file format. Especially, if the data volume is increasing this kind of database performs much better in comparison to relational databases. In other words, a round robin database improves the scalability and growth of the NoAH architecture. For example, the NFdump[15] tool uses this kind of database. In addition, tools like NFsen[16] already exists that support the analysis of the Netflow data by a graphical presentation and filter functions.

8.2.5 Alert data

An alert is generated if an attack has been detected by the different detectors in the NoAH framework. For the detection of attacks we refer to deliverable D1.2. A rough orientation for the structure is given by the IDMEF XML format[10] intended to be used as an exchange format for alerts generated by intrusion detection systems. We propose that the alert data should cover at least the following information derived from the IDMEF format:

- Attack identity
- Detection time
- Creation time of alert
- Source of attack
- Target of attack (honeypot Identity)
- Identity of site which owns the attack data
- Identities of corresponding signatures
- Vulnerability identifiers
- TCP/IP Packet data

In contrast to the Netflow data which is collected by network sensors (e.g. NFdump) the network packet data is captured on high-interaction honeypots. Because all known attacks can be filtered in the first stage, the extent of the packet data can be expected to be less in comparison with the Netflow data, although it contains the full payload of the packets. Therefore, we propose to store the data in the relational database. If it turns out that the data volume is higher than expected the TCP/IP packet data can be stored alternatively in a round robin database.

It should be kept in mind that statistical data about attacks will be presented on the public part of the NoAH web server which must not include confidential information maintained in this part of the NoAH database. Therefore, these information have to split into a confidential part containing the sources and targets of the attacks and the packet payload and a non-confidential part. This will allow to configure a database user only being granted read access to the non-confidential parts of the alert information.

Another problem arises if relays and high-interaction honeypots are located in a private subnet or if the data is sent through an anonymizing proxy (e.g. Tor). In this case, the true IP address of the attacker is not directly known and has to be passed to the sensor by another way.

8.2.6 Signatures and related information

To generate signatures for novel attacks is one of the goals of the NoAH project. As proposed in deliverable D1.2 multiple types of signatures covering network based and/or host based signatures may be generated. Therefore we propose to distinguish between these different types for the specification of the database scheme.

- Vulnerability component
  - Identity of vulnerability
  - Identities of other sources e.g. CVE Mitre, CERT/CC, Securityfocus
  - Affected service and operating system specification
  - Signature IDs
- Signature header
  - Signature ID
  - Reference to signature data
  - Identity of vulnerability
  - Identities of corresponding alert
- Signature data (host-based)
- Signature data (network-based)

In analogy to the Netflow data, the signature data can be split into a confidential and non-confidential part if decided by the NoAH consortium.

8.2.7 Database Access

In this section the interfaces that allow access to the database are described. Concerning the access we distinguish between the administrative, operational, and user role. For each role a corresponding database user can be configured only being

granted the privileges needed for accessing the necessary data (minimum privilege principle). Additional database users are configured having read-only database access. These users are required for accessing the data presented on the public and private parts of the NoAH web server. The minimum privilege principle limits the possible abuse in the unlikely case an attacker gains unauthorized access to the database. For example, an attacker should only have very limited read-only database access if she was able to compromise the NoAH web server. This prevents the attacker from having access to critical private data (e.g. the IP addresses of the honeypots) concerning the NoAH architecture.

Access to the NoAH database is given by the following interfaces:

Public interface. NoAH results will be provided on the public part of the web server. Accessing this data will be done by the ”public interface” for which a ”public” database user is configured . As mentioned above abuse can be effectively limited by only granting read access for this user to the non-confidential parts of the NoAH database.

Semi-Public interface. It seems to be advantageous to distinguish between a public and semi-public data. Semi-public data can also cover information about unknown attacks or new signatures that should not be provided to completely untrusted sites. For example, this data can be provided to CSIRTs and security companies to react to these novel attacks or vulnerabilities. Access to this data is given by a semi-public interface.

NoAH partner interface. The NoAH database contains confidential data that belongs to different NoAH partners. For example, the NoAH policy may grant sites access to all data resulting from their honeypots. However, access to confidential data resulting from honeypots of different sites may be prohibited. For example, this data will include the source and target IP addresses in monitored attacks. To cope with these privacy issues, this interface have to authenticate the invoking site first and then apply the anonymization or pseudonymization filters according to the site identity. For example, this can be done by restricting access only to authenticated sites (e.g. by a S/MIME client certificate) and evaluating the site identity by the certificate data. If the identity of the invoking site is known all confidential data belonging to other sites can be anonymized or pseusonymized before the export of this data.

In addition to the above mentioned interfaces, the administrative and operational roles need access to the database. Corresponding to both roles a database user can be configured having read and write access to the required tables of the database. With exception of database component dedicated to the configuration and status of the honeypots there is no overlap concerning the access to the tables. In other words, access to the tables needs only to be granted to either of these roles. For the previously mentioned honeypot data, we propose to grant access to both roles.

Chapter 9

Summary and concluding remarks

In this document, we described the various interfaces between components forming the NoAH architecture. Specifically, we identified the communication requirements of each component and we examined the interface between low- and high-interaction honeypots, external components (honey@home and funneling/tunneling) and core honeypots and finally high-interaction honeypots and signature generation components. Low-interaction honeypots – mainly honeyd – handoff communications to high-interaction honeypots using normal TCP connections, thus no modification is needed to high-interaction honeypots and also functionality exists in honeyd. For the interface between honey@home and funneling/tunneling with the NoAH core, an additional component is required. This component, called connection daemon, accepts incoming connections from honey@home clients, unwraps encapsulated packets and injects them to low-interaction honeypots. Signature generation components take exported data from Argos containment environment as input to generate new alerts. Finally, the distributed infrastructure of NoAH requires an administrative interface to people and organizations that are involved in the project, and also an interface to people that want to participate. We examined the design of NoAH database, that includes information about honeypot configurations, data collected and contact information.

Bibliography

[1] Bochs ia-32 emulator project. *http://bochs.sourceforge.net/*.

[2] Carmentis - a german early warning system. *http://www.cert-verbund.de/carmentis/index.html*.

[3] Common model of system information (cmsi). *http://www.cert-verbund.de/cmsi/en.html*.

[4] D0.2: Requirements collection and analysis. *http://www.fp6-noah.org/publications/deliverables/D0.2.pdf*.

[5] D1.1: Honeypot node architecture. *http://www.fp6-noah.org/publications/deliverables/D1.1.pdf*.

[6] Google hack honeypot. *http://ghh.sourceforge.net/*.

[7] Honeybee. *http://www.thomas-apel.de/honeybee*.

[8] Honeybot. *http://www.atomicsoftwaresolutions.com/honeybot.php*.

[9] Honeynet project. *http://www.honeynet.org*.

[10] The intrusion detection message exchange format. *http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-16.txt*.

[11] Kfsensor. *http://www.keyfocus.net/kfsensor*.

[12] Labrea. *http://labrea.sourceforge.net/labrea-info.html*.

[13] Leurre.com honeypot project. *http://www.leurrecom.org/* .

[14] Mwcollect. *http://www.mwcollect.org* .

[15] Nfdump. *http://nfdump.sourceforge.net/*.

[16] Nfsen - netflow sensor. *http://nfsen.sourceforge.net/*.

[17] Qemu homepage. *http://fabrice.bellard.free.fr/qemu/about.html*.

[18] Sdl library.

[19] Sebek homepage. *http://www.honeynet.org/tools/sebek/*.

[20] Systrace - interactive policy generation for system calls. *http://www.citi.umich.edu/u/provos/systrace/*.

[21] User-mode linux. *http://user-mode-linux.sourceforge.net/*.

[22] Vmware homepage. *http://www.vmware.com/* .

[23] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and N. Provos. A hybrid honeypot architecture for scalable network monitoring. In *CSE-TR-499-04*.

[24] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX 2005 Annual Technical Conference*, 2005.

[25] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levin, and H. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of the Recent Advance in Intrusion Detection (RAID) Conference 2004*, September 2004.

[26] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[27] N. Provos. A virtual honeypot framework. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, August 2003.

[28] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[29] D. X. Xuxian Jiang. Collapsar: A vm-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, pages 15–28, August 2004.