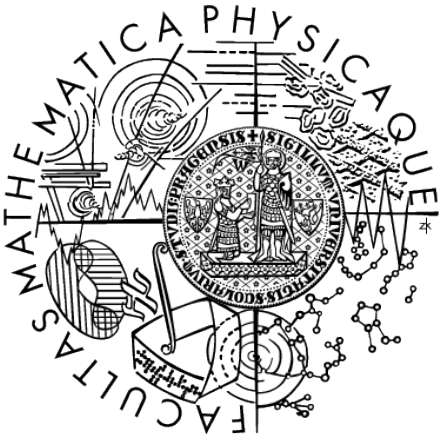
Faculty of Mathematics and Physics
Charles University in Prague

Master Thesis



Tomáš Kouba

Virtual honeynet with simulated user activity

Department of Software Engineering

Supervisor: Mgr. Pavel Kaňkovský

Study programme: Computer Science

I would like to thank my supervisor Mgr. Pavel Kaňkovský for all the pieces of advice and great patience.

Thanks to my colleagues from IRC channel #twin and especially Jiří Kosina, Eduard Bejček and Miroslav Beneš for the moral support.

I hereby declare that I have written the diploma thesis myself, using only cited sources. I agree with lending and distribution of the thesis.

Prague, 12th December 2008

Tomáš Kouba

Title: Virtual honeynet with simulated user activity

Author: Tomáš Kouba

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Kaňkovský

Supervisor's e-mail address: peak@mbox.troja.mff.cuni.cz

Abstract: The goal of the work is to design and implement a honeypot (a trap for attackers) that will be able to simulate working user and other usual system activity in a convincing way so as to make it difficult to distinguish a honeypot from an ordinary system, will keep a stealth record of actions of any attackers who would attack the honeypot, and will make it possible to deploy a whole virtual network of honeypots (a honeynet) on a single host machine. The implementation should be resistant to any of the well-known techniques used to detect a modified operating system or OS kernel such as the kstat utility.

Keywords: Honeypot, stealth techniques, rootkit.

Název práce: Virtuální "honeynet" se simulovanou uživatelskou aktivitou

Autor: Tomáš Kouba

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Pavel Kaňkovský

e-mail vedoucího: peak@mbox.troja.mff.cuni.cz

Abstrakt: Cílem práce je navrhnout a implementovat "honeypot" (past na útočníky), který bude umět simulovat uživatelskou a jinou obvyklou systémovou aktivitu dostatečně přesvedčivým způsobem tak, aby ztížil rozpoznání honeypotu od obyčejného systému. Bude také vést skryté záznamy o akcích útočníků, kteří honeypot napadnou. Bude umožňovat sestavit celou virtuální síť těchto honeypotů (tzv. honeynet) na jednom fyzickém stroji. Implementace by měla být odolná vůči známým technikám používaným k detekci modifikovaného operačního systému nebo jeho jádra jako to delá např. utilita kstat.

Kličová slova: Honeypot, skrývací techiky, rootkit.

Contents

Chapter 1

Introduction

1.1 From history to present times

First years of the Internet were characterized by enthusiasm for new possibilities and main effort was put to its functionality. Security aspects were turned aside and computer security was reserved for military. As more and more threats arose in nineties (e.g. automatic attacks, script kiddies) security issues became much more important for computer science and some formal techniques, protocols and common habits were established.

Various research groups and projects focused their attention on security. One great aspect of protecting a network or one computer is studying techniques and steps performed by an attacker (considering human attacker and also automatic worms). This can be done by monitoring public mailing lists and web pages of *black hats*[1]. To name one as an example we can mention Phrack[2]. But not all attacker's finesses and used tools are published and those unpublished might be the most dangerous ones.

1.2 A pot full of honey

Systems called *honeypots* have been developed for catching attackers and for studying their actions on the compromised system. The term *honeypot* is used for a system that has been configured with intention to be compromised (so it usually contains older software with security vulnerabilities) and to get information about attacker's techniques and tools she uses. There are various ways how to modify a system to log all attacker's activity and these approaches to building honeypots are discussed in this thesis in Chapter 2.

The next level of honeypots is a whole network of honeypots or so-called *honeynet*. It is built on the idea that an attacker is usually trying to compromise the whole network

[1] Black hat is a term used for a security expert with malicious intentions. The experts on the other side of the computer security battlefield are called *white hats*.

[2] http://www.phrack.com/

of the attacked company. We discuss Sebek [33], the most popular project used for building honeynets, in Section 2.4.

The ideal honeynet solution for this thesis would provide the following features:

1. watch and log attacker's activities
2. deployable on a virtual infrastructure
3. difficult to detect

All these aspects with theory involved in honeynets are also discussed in Chapter 2.

When building a virtual honeynet there are other important decisions that have to be made:

1. what virtualization product (or implementation) should be used
2. what techniques will be probably used by the attacker, how will we react on them and how will we monitor these techniques so we can study them

1.3 Layout of the thesis

As we have already said, the Chapter 2 describes honeypot technologies and theory. The Chapter 3 describes techniques how an operating system (we focus on Linux only) can be modified and mainly how this modification can be hidden so even a user with administrator privileges cannot discover it.

In Chapter 4 we discuss virtualization theory and compare various virtualization software from many points of view (e.g. speed, completeness, approach used for virtualization). At the end of the chapter we decide what software we will use.

In Chapter 5 of the thesis we build a *virtual honeynet*. It is a network of virtual computers, whose operating systems are configured to become honeypots. All these virtual computers are running on one physical machine. The same chapter also describes library *VAccess* that has been developed for this thesis. It allows honeynet administrator to monitor the activities and processes on one particular honeypot.

To mimic a real system we have implemented a simple user simulator which is described in Section 5.2.

Chapter 2

Honeynets

2.1 Definition

The term **honeypot** has no exact definition used world-wide. In this thesis we will use the following definition of honeypot: *"A server that is configured to detect an intruder by mirroring a real production system. It appears as an ordinary server performing usual work. The honeypot is used to learn about an intruder's techniques as well as determine vulnerabilities in the real system."*

Similarly honeynet is then "*A network consisting of honeypots. A 'virtual honeynet' is one that resides in a single physical server, but pretends to be a full network."*

The main tasks of every honeynet are:

- log attacker's activity
- capture used tools
- do not reveal the presence of itself to the attacker

Additional useful features:

- do not allow an attacker to misuse honeynet's resources for further attacks
- alert the administrator of the honeynet automatically about every intrusion (work as an IDS)

2.2 Taxonomy

The honeypots can be divided into 3 categories:

- logging-only honeypot
- low-interaction honeypot
- high-interaction honeypot

Logging-only honeypot is a machine without any special software. All traffic from and to the machine is logged and consequently analyzed. It does not implement any decoy that should attract an attacker. The honeypot itself does not distinguish between successful and failed attack. It merely logs related data. This type of honeypot does not interact with a user, it usually just records all the incoming packets for later analysis and does not respond to these packets. It is mainly used for portscan[1] detection.

Low-interaction honeypot is a machine with special software which mimics behaviour of standard network services (e.g. web server, SNMP service, FTP server) and captures the malicious data issued by an attacker against these services. The efficiency of such honeypot depends on the ability of the service to implement a significant subset of the given protocol (e.g. HTTP protocol) and simulate responses of the real service (e.g. Apache web server). Low-interaction honeypots are typically used for following purposes:

- Statistical analysis of well-known attacks. It means we know that attackers are aware of some vulnerability and they exploit it. And we try to get sources and/or counts of the attacks in time.
- 0day[2] vulnerabilities. We can filter all known attacks from the honeypot and if there is still some network communication with suspicious footprints, it is highly probable that we are facing an attack against a freshly discovered vulnerability.

High-interaction honeypot is a machine with standard software suite installed. Administrator of such honeypot usually installs vulnerable versions of network services so it allows attacker to break in the machine, install malicious tools and complete the attack. Meanwhile the attacker is watched and all her activity is logged and used tools are stored for later analysis. This kind of honeypot becomes very useful because logging-only is not sufficient anymore. Attackers started using encryption (mostly SSH) and so their actions must be logged after the SSH packets are decrypted (on the target machine).

The research and development of honeynets is not concentrated on one place and many projects are active on this field. In [26] there is a good overview of existing honeynet projects with analysis of their:

- architecture
- communication between honeypots and other parts of honeynet
- data capture
- containment strategy (how the project protects the world outside when the attacker is successful in compromising the honeypot)
- data analysis
- effort for deployment and supervision (how difficult is to install the project and maintain it over time)

An ideal software implementing honeypot would have the following features:

[1] Portscan is a procedure performed by the attacker before the actual attack on the target. It usually consists of finding out what IP ports are open and what software (in what version) is running on the target.

[2] 0day is a popular abbreviation of *zero day* and it means that the given fact (most often a vulnerability) has not been published and described yet.

1. Completely log all activities, store transfered binaries, log commands issued by an attacker.
2. Be undetectable by an attacker (it must be indistinguishable whether the system has the honeypot turned on or off).
3. Do not consume too much resources so the system is still usable.

2.3 Caveats

Honeypots became widely popular for gaining forensic data and for an analysis of attacks. But attackers are aware of this fact and they have developed several methods how to circumvent honeypot and do the malicious activity without being logged or with logs being useless. The most popular steps to do when an attacker is not sure about her privacy are:

- binary encryption
- userland exec
- stealthing

See chapter 3 for more details on how rootkit authors fight against forensics analysts or virtualized environment.

2.4 Sebek

Currently the state of the art in honeynet area is a technology called Sebek [33]. It can log almost all actions performed by a user on the honeypot and send the record to a central collecting server. The main unique features of Sebek are:

- records all text typed in a terminal
- saves all tools transfered from or to the host even if they are deleted afterwards
- records all binaries executed on the host even if they are deleted afterwards
- can filter what should be recorded (so the analysts does not get drowned in the log files)
- hides the network activity related to Sebek from all users on the network[3]

2.4.1 Architecture

Sebek is a tool with client-server architecture that allows the administrator to monitor relevant activity on several nodes in his local network. The way Sebek works is shown on figure 2.1.

[3]This applies under condition that all nodes on the network are running an instance of Sebek.

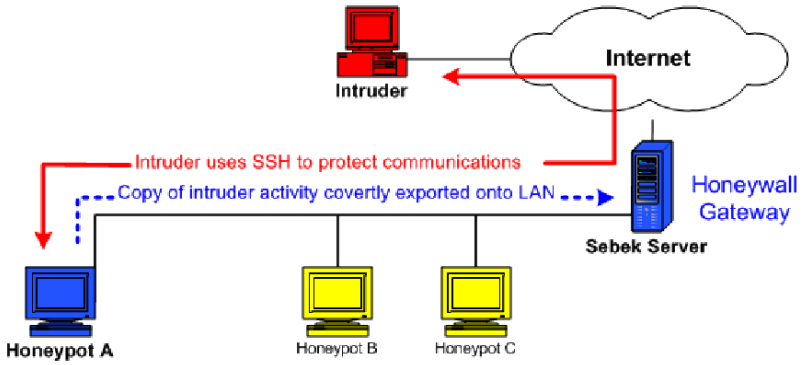


Figure 2.1: Sebek architecture

The whole technology is divided into several parts:

- client – It is available in two forms: kernel module or kernel patch. The goal of the client is to capture all relevant data from the node where the client runs and send them to the Sebek server. Sebek itself and data sent must be hidden from root on the node and also from all other nodes on the local network except Sebek server.
- server
  - sbk_extract is a low level tool that collects the data from UDP stream and dumps it to standard output
  - sbk_ks_log.pl, sbk_diag.pl, sebekd.pl are high level tools that process client messages and present them to user in readable form or dump them to a database for further analysis.

2.4.2 Disadvantages

The main disadvantage of the Sebek is that it is detectable by design. The attacker who has gained the root access can load its own kernel module and detect or even disable Sebek. She can also boot another kernel (although it is a suspicious step usually discovered by an administrator).

The problem is the fact that Sebek is a part of the running kernel. If compiled as a kernel module it can be discovered by an attacker by methods to reveal hidden kernel modules (described in 3.3). These are usually used by white hats to check for hidden kernel modules installed by black hats but this time even the black hat can use such methods.

Approaches for revealing Sebek specifically are presented in [7]. The main ideas presented in [7] are based on comparison of two data sets: the first is retrieved inside the suspicious kernel and the second outside of the suspicious kernel. Since Sebek filters the network traffic going from the machine, some network counters (e.g. transferred bytes counter) are different in the mentioned data sets.

The paper [7] also presents methods how to remove Sebek from running kernel or circumvent its ability to keep forensic logs. In future work section the authors also show some ways how to harden Sebek against their own algorithms.

But in general principle, Sebek runs in the kernel that is being attacked and so it can be revealed by the attacker that has full access to the kernel.[4] Another problem is that the attacker can study sources of Sebek and find some common patterns in its code. As long as the attacker is able to inspect kernel memory[5], she can look for the common patterns of Sebek and so has quite good chance to reveal it.

2.5 Honeypot conclusion

Although the Sebek is the honeypot "state of the art" it does not fulfill the requirements presented at the end of section 2.2 because it can be quite easily detected. The main flaw is that the Sebek is implemented inside the honeypot's kernel and cannot be hidden completely by design.

In chapter 5, we will present our honeypot implementation that tries to overcome this disadvantage.

[4] This is true in default kernel where there are no limitations on root account.

[5] This inspection can be done in various ways: by kernel module, by reading /dev/kmem, by reading /dev/mem etc.

Chapter 3

Stealthing and anti-stealthing

The main feature of successful honeypot is that the attacker (which has already gained the root access) cannot find out that the system is watching her, logging her activities and keeping her tools for future analysis. Generally we need to run a kernel with special abilities that look like an ordinary kernel.

Honeypot is not the first area which demands this kernel behaviour. *Rootkits* have very similar needs. Rootkits are special software used by attackers (see section 3.1 for detailed description). Although rootkits are oriented on modifying running kernel (while we are able to change the source code of the kernel), the techniques used to discover rootkit can be used against our honeypot too.

The first sections of this chapter will describe rootkits, where *stealthing* is used to hide the fact that kernel has been modified. *Antistealthing* is then used by white hats to check the integrity of the kernel or the whole system. At the end of the chapter we will make conclusions about how stealthing and antistealthing is important and useful for honeypots and honeynets.

3.1 Rootkits

3.1.1 Rootkit definition

The term rootkit is widely used but not precisely specified. For this thesis we will stick to the Wikipedia definition[52] which says: "*A rootkit is a general description of a set of programs which work to subvert control of an operating system from its legitimate operators*". Such programs are usually used right after successful attack on a system. Attacker installs the rootkit so she can easily return and gain privileged rights lately again.

The primary function of a rootkit is to allow its mistress to regain the control over the attacked machine. However, modern rootkits have many more features:

- purify logs from traces of malicious activity
- hide presence of suspicious system objects (files, processes, open sockets, kernel modules, etc.)
- obtain user's passwords (so called sniffing)

3.1.2 Rootkits taxonomy

During past few years, black hats developed several rootkits and used various techniqes and system levels to implement them. We can divide rootkits into several categories according to the way they modify the system:

- user level utilities (section 3.1.2.1)
- loadable kernel modules (section 3.1.2.2)
- kernel memory modifications (section 3.1.2.3)

3.1.2.1 User level utilities

The main idea of this approach is to replace important system utilities by modified ones. The most important utilities are `ls, ps, netstat, syslogd` etc. This kind of rootkit can be simply discovered by comparing output of modified utility with correct output[1]. The binary of the utility itself can be checked too (or rather its hash because of size issues).

The creating database of hash values and regular checks of binaries can be done automatically and it is a goal of various projects (e.g. Tripwire [18] or Sentinel [47]). These applications are called *integrity checkers.* Their usage is of course limited to the case when we know that the checker is not modified itself.

The examples of user level rootkits are:

- T0rn
- lrk3
- Ambient's Rootkit

3.1.2.2 Loadable kernel modules

All user level rootkits suffer from the fact that the administrator can compare information provided by the modified utilities to the actual output from kernel. This means that he can for example compare content of `/proc` directory with the output of suspicious `ps`. Or he can keep his own statically linked `ps` on USB token and compare the output from this `ps` and the `ps` in the system.

So the next step made by rootkit authors is changing the data that kernel publishes to userland. The most convenient way to change or enhance Linux kernel's behaviour is Loadable Kernel Module (LKM).

Typical rootkit kernel module wants to hide:

- Presence of itself — This can be easily done by removing its record from the list of all modules.
- Presence of certain files and processes — This is typically done by changing semantics of system calls *getdents* and *getdents64*.

[1] The correct output can be retrieved from original utility taken from reliable source (e.g. external media).

To change the system call semantics the attackers used to rewrite the address of given system call in syscall table with address of her own malicious function. Since kernel 2.6 the syscall table is not exported and the attacker cannot simply rewrite its content. First the table has to be located in the memory. This is typically done by searching kernel memory for known patterns[2].

However, recent kernels even place the table in read only page and so the attempt to rewrite its content results in kernel oops. Under these circumstances kernel rootkits usually copy whole table and rewrite the address of the table in assembly code. The places that must be changed are those where kernel computes the address of a syscall when it is invoked. Namely those are kernel functions *sysenter_entry* and *system_call.*

The example of a simple rootkit with sources can be found on the enclosed CD in `sources/hide_proc`.

Examples of LKM rootkits:

- knark
- adore
- rial
- rkit

3.1.2.3 Kernel memory modifications

LKM rootkits became a serious threat for system administrators because "how can you trust a system when you cannot trust its kernel?" But there was a piece of advice: Disable support for loadable modules in your kernels.

Yes, Linux kernel has the compile option that allows administrator to build kernel without modules. All features are then compiled in the main kernel image and loaded during the boot sequence. This compile options has been implemented mainly for the sake of kernel size[3] and now it used also for security reasons.

But attackers found their way to break into kernel again. Silvio Cesare in [3] presents method how to inject code into running kernel without using LKM. In [39] authors present solution for many small problems that must be solved when the attacker actually wants to inject her code into kernel that lacks module support. We describe the details of this method in the appendix C.

3.2 Attacker's anti-forensics steps

3.2.1 Common attack scenario

When a machine is attacked and attacker wants to completely control the host (wants the root access), she has to pass protection barriers that are quite similar on all systems.

[2]Known pattern can be address of some exported symbol like for example *sys_read.* The addresses of these symbols can be obtained from /proc/kallsyms for running kernel or they are usually installed by Linux distributors in /boot/System.map-<kernelversion>.

[3]Kernel without modules does not have functions for loading them, relocating symbols etc.

That is why many attacks have a common scenario. Attacker usually has to do the following steps:

- Scan the ports of the attacked machine for listening applications (e.g. HTTP server Apache).
- Check the version of listening applications according to their network protocol (e.g. apache in default configuration tells its version on every error page).
- If there is a known vulnerability of one of these applications, use an exploit to gain remote access.
- The gained access has privileges of the user running the vulnerable application (e.g. Apache runs usually under a user called apache).
- **Launch a program that gains root privileges** for attacker (this is usually done by exploiting vulnerability in suid executable or kernel).
- Install rootkit.
- Remove traces of the attack (delete exploit and rootkit binaries, clean up the logs or make them unusable).
- Do malicious actions (e.g. scan other machines, send spam, steal user files).

Every mentioned step can be a problem for a paranoid attacker. If the administrator is monitoring the network or even have installed Intrusion Detection System (IDS), the attacker cannot be sure if her attack remains unnoticed.

We will focus on the emphasized point because transferring the malicious executables, launching and removing them can be the moment where honeypot can become useful (see section 3.4 for further discussion).

If the attacker is watched by the system administrator (or some IDS), caught in action and disconnected, the used program can be analyzed by the administrator and the exploited vulnerability can be revealed and fixed. Attackers don't want their tools to be analyzed and they want to use them as long as possible. We will present two countermeasures used by attackers. First will be *executable file encryption* (section 3.2.2) and second *avoiding exec()* (section 3.2.3).

3.2.2 Executable file encryption

The first step attackers take when protecting their executable files is encrypting them. This does not protect them from being found and being the evidence that something suspicious is happening. But the encryption makes the forensics process much harder for a system administrator.

The executable file encryption means either:

- Encryption on disk and decrypting by hand when the executable file is to be used.
- Modifying the executable (usually stored on disk in ELF[4]) so it asks for password and decrypts itself automatically when launched.

[4]ELF (Executable and Linkable Format) is a standard file format for executables.

The latter option is much safer because it ensures that there is no time period in which the binary would be on disk unencrypted. The encryption can be integrated in the program itself but it is better to use generic ELF encryptor like for example **burneye**, **dacryfile** (presented in [15]) or **shiva** [23].

These encryptors use similar techniques as so-called *elf viruses*. They find a gap in one of the elf segments (there are gaps because the segments are mapped directly to memory during loading and so they must be aligned on page boundary). The encryptor puts the decrypting code in this gap and redirects ELF entrypoint (*e_entry* in ELF header) to it. The decrypting code usually asks for password and decrypts the other loaded ELF parts. This technique is used in the mentioned dacryfile utility.

More modern approach for encryptor is to create a stub that does the whole loading and relocation of encrypted ELF file. This is very similar to so-called *userspace exec* which is described in the following section. Basically the ELF is wrapped into another ELF that decrypts the executable file and loads it into memory without help of underlying kernel (which is the one who normally loads and executes programs). This allows the decryptor to be more complex because it is not limited by the average size of gap in ELF segments.

3.2.3 Execve() and avoiding it

When an attacker needs to execute some exploit she can do it in a normal way like any other executable program. This is done by calling system call *execve().* This call then tells kernel to load the binary executable file from disk, prepare memory layout for the program, load additional libraries (with help of dynamic linker) and launch the program. And, this can be the problem for a paranoid attacker. The *execve()* call needs to load the binary from disk and it means that there is some time when the administrator (via some automatic script or manually) can copy the binary and analyze it.

The solution for this problem is to avoid the *execve()* system call and do all the kernel work in user space. If one executable would be able to launch another binary executable without loading it from a disk, it can transfer it for example via secure channel from the Internet and so minimize the local administrator's chance to analyze the executable and discover what vulnerability has been exploited.

This approach was first presented in [14]. The article describes basics of ELF binary format and how kernel loads an ELF binary in step-by-step manner. It shows what steps have to be done by the user space implementation and the list of problems that must be solved. Finally, a proof of concept code *ul_exec* is enclosed in the article.

In order to load and execute a binary executable the *ul_exec* has to do the following:

1. Store the parameters passed to main() in argv because they will be overwritten in next steps.
2. Clean the address space of the launching process. This means unmapping the areas of virtual memory that will be occupied by the new binary (in fact only .text, .data and .bss sections are needed, others will be loaded by dynamic loader – see below). This is why *ul_exec* is implemented as a shared library. The unmapping code does not belong to the main .text section and so it does not cause segmenation fault.

3. Load dynamic linker if the binary is dynamically linked executable. The dynamically linked ELF executable contains one segment PT_INTERP which defines the path to a program that should be used to start the main executable. In typical Linux case the string is "/lib/ld-linux.so.2".
4. Load the main binary. This means simply loading PT_LOAD segments from ELF to appropriate memory addresses.
5. Initialize the stack.
6. Determine the entry point. Entry point is determined in ELF header as an address in virtual memory. For binaries using shared libraries the entry point is the start of dynamic linker (which resolves symbols, loads libraries and starts the application). For static binaries the entry point is the start of the application itself.
7. Start the new loaded binary. This is done simply by jumping on the entry point determined in step 6.

3.3 Identifying patched kernel

Usual LKM rootkit as described in section 3.1.2.2 hides itself from the list of kernel modules. It is typically done in module's initialization routine by unlinking the module from the global kernel list of all modules. This section presents some techniques how such hidden modules can be revealed. We also name software which implements these techniques.

3.3.1 Checking within the operating system

There are several projects trying to check if the running operating system has been compromised and modified:

3.3.1.1 Chkrootkit

Chkrootkit *"is a tool to locally check for signs of a rootkit*" [25]. It consists of several binaries and one big shell script. When the script is executed it checks system binaries for changes, looks for promiscuity of network interfaces, checks if system log files (e.g. wtmp) have been changed and searches disk for traces of well-known rootkits.

The checks performed by chkrootkit are great for revealing rootkits used out of the box. It means that the attacker just downloads them from author's site and does not modify their behaviour (e.g. the location of hidden files or charactertic log entries). On the other hand, these checks are weak against new rootkits. One of the weakness is that the checks rely completely on standard OS interfaces.

3.3.1.2 Kstat

Kstat[10] is an utility suitable for kernel 2.4 According to the author *"It sports network socket dumps, sys_call fingerprinting, stealth modules scanning and more."* The kstat has many options and modules that check various aspects of the system:

- Dumps info about process. The info is retrieved right from `/dev/kmem` not procfs file system.
- It lists the loadable kernel modules. The kstat first goes through the kernel internal list of modules and dumps them. Then, it scans whole kernel memory for module-like structures. The first approach reveals modules that are hidden by modified query_module system call or /proc/modules file. The later approach is more time consuming but it can reveal even module that unlink itself from the internal list.
- Kstat can store integrity data (addresses in syscall table, IDT handler, etc.) at the moment when the administrator knows that everything is all right. Then, these data can be used by kstat to check current status and reveal malicious modification.

3.3.1.3 Zeppoo

Zeppoo is very young project similar to kstat. The author describes it: "*Zeppoo allows you to detect rootkits on the i386 architecture under Linux by using /dev/kmem and /dev/mem. It can also detect hidden tasks, modules, syscalls, some corrupted symbols, and hidden connections.*"[53]. It is written in python using its own library libZeppoo to inspect `/dev/kmem` or `/dev/mem`.

The tool is typically used in two steps:

1. Create a fingerprint of clean system (this has to be done for every kernel update).
2. Check regularly if the fingerprint corresponds to current state of the system.

The main advantage of Zeppoo is that it works even on `/dev/mem` device and so `/dev/kmem` can be omitted (useful for Red Hat kernels, where /dev/kmem is usually switched off).

3.3.1.4 Other detection tools

There are other projects for rootkit detection. Many of them are mentioned and analyzed in [29]. The principles used for the detection are the same as in kstat, zeppoo and chkrootkit so we just list then without further description: St Michael, Carbonite, Samhain, checkps, Rkscan, RootCheck, Rootkit Hunter.

Although these methods can be more and more sophisticated in the future, the fact is that they rely on information given by system which is controlled by the attacker so the negative result of searching for kernel modification does not mean it is not modified.

3.3.2 Hardware checks

As we already said the modified kernel cannot be trusted. But we still want to read the memory and analyze it. This can be done by an external device on PCI bus that would read and transfer the memory to the trusted host or read and analyze it itself.

The existing architecture achieving this goal is presented in article by Petroni et al. [29]. The architecture schema (figure 3.1) consists of special PCI card inserted in the

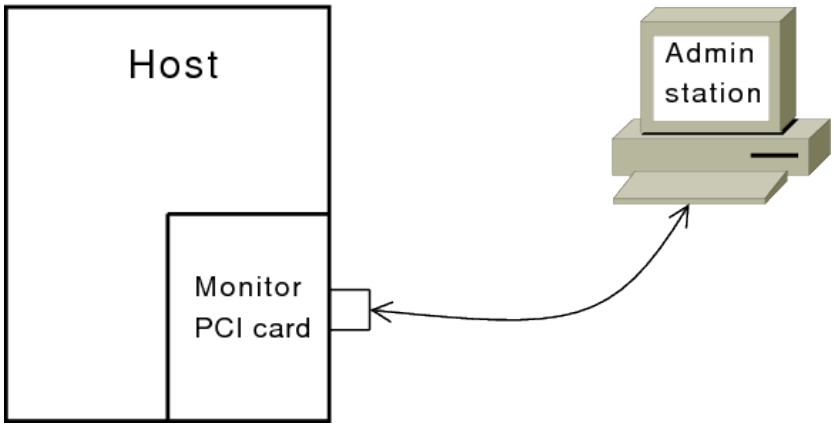


Figure 3.1: Copilot architecture schema

monitored *host* and connected via *independent communication link* to so-called *admin station*.

The integrity monitoring is done by regular hashing of important parts of kernel memory. The hash is computed on the admin station from host's kernel memory read by monitor PCI card. The card uses DMA to read the memory so the host CPU is not involved in this operation.

If the hash differs the administrator is alarmed. The question is what parts of kernel memory are important and should be monitored for integrity (hashed). The first would be text (code) section of the kernel to ensure that the code has not been modified. But it is not enough. The authors of [29] admit that there are many jump tables in kernel (first would be *sys_call_table*) that has to be checked and making a complete list of these tables is a big challenge. The article focuses on proof of concept and shows that in the case of *sys*_call_table (which is misused by attackers in most rootkits) their approach works well.

The rest of the article discusses host performance (which is not significantly decreased), security (attacker can block the PCI card but this action is monitored by admin station) and reliability (it detects all discussed rootkits).

One of the main advantages of this approach is the ability to detect even new rootkits. The main disadvantage is the need for a special hardware and dedicated admin station. So, although this technique is very successful, it is hard to use it in a production environment and it currently stays in academic labs.

3.4 Conclusions for honeypot

This chapter showed that the fight between the attacker and the administrator is similar to "cat vs. mouse" game. Every time the attacker comes with a new technique how to hide herself, the administrator deploys his new technique to reveal the attacker. Then the attacker develops another approach and the game continues.

In general principle administrators are one step before attackers. This is because administrators install the software and can take any precautions to harden their software

and to prepare honeypot-like traps to detect malicious activity or spy on attackers.[5] They can also install special hardware as was described in Section 3.3.2.

The similar approach can be used when the hardware is virtualized. The memory can be checked and interpreted "from outside" in a similar manner. In the next chapter we describe various virtualization solutions and choose one of them. The honeypot built on this virtualization solution with monitoring software is presented in the Chapter 5.

[5] On the other hand many systems are installed with just default security configuration.

Chapter 4

Virtualization

Honeynets currently deployed by projects like SEBEK are aimed at a real network of physical computers. This approach needs lot of resources and is quite difficult to manage. Another problem is collecting data via information channels that should be hidden from a attacker. Hiding such channel is non trivial task.

Virtualization solves both of these. Real systems can be simulated on one physical machine. It is cheap and easily manageable. Building the hidden channel is also easier because it can be incorporated in the virtualization infrastructure. Then the channel would not be detectable in the guest system.

First we will describe what virtualization is and what approaches can be used to achieve the state where many machines run on one real hardware.

4.1 Virtualization theory

In [40] Amit Singh defines virtualization: *"Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.*"

This definition encapsulates even things that we wouldn't call virtualization nowadays: multiprogramming, multitasking, multiuser enviromnent. Virtualization term that will be used in this chapter should be understood as a complex execution enviromnent (in the sense of previous definition) that resembles real physical machine in a convincing way as much as possible.

4.1.1 VMM

The term *VMM* stands for Virtual Machine Monitor and it has been introduced by Popek and Goldberg in [31]. VMM is a layer that communicates itself with hardware and provides an abstract hardware that underlaying operating system (often called guest OS) communicates with. The concept is shown in the figure 4.1

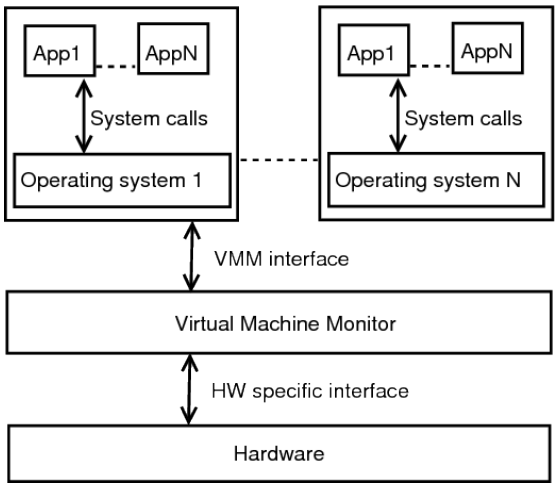


Figure 4.1: VMM concept

In an ideal situation the guest operating systems would not be able to distinguish whether they run on a real physical hardware or on an abstract hardware layer provided by VMM.

4.1.2 Emulation

Virtualization.info[48] defines emulation as *"A software technology allowing an operating system or an application to run on hardware other than the one for which it was developed."*. In this thesis we will call emulation the virtualization approach in which every instruction is emulated (interpreted) by the emulator. The control mentioned earlier can be done in various ways and on various levels. For example Bochs is a true emulator of x86 architecture since it gets every instruction and interprets it. On the other hand VMware or QEMU (with KQEMU accelerator) uses the fact that they run on the same architecture they emulate and so they let short segments of machine code to be executed right by the CPU (for caveats of this approach on x86 see Section 4.1.5). The important differences from other approaches (like paravirtualization – Section 4.1.3) are:

- emulators provide virtual devices that try to mimic a real hardware (not only an idealized hardware)
- they emulate other parts of the architecture (e.g. BIOS on x86).

In some articles emulators are also called CSIMs (complete software interpreter machines). The VMM for the emulation case is the emulator itself.

Example projects using emulation are: Bochs[43], Plex86[44], VMware[19] or QEMU[1].

4.1.3 Paravirtualization

Paravirtualization as a scientific term is not very well defined. Many sources differ in its definition and meaning but they agree in one aspect: Paravirtualization is a virtualiza-

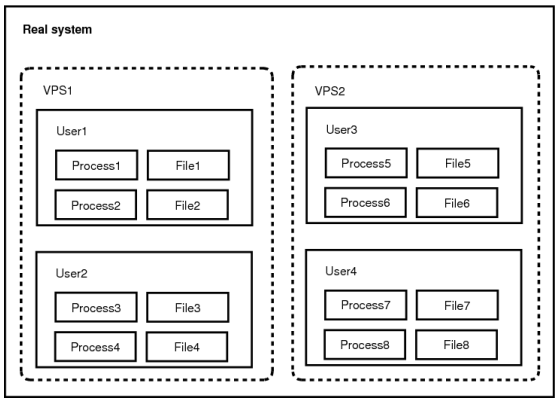


Figure 4.2: Additional grouping level.

tion technique that requires guest OS to be prepared for virtualization. This means that a Linux kernel have to be modified to be able to run as a guest OS in a paravirtualized environment. This OS preparation is done for 2 reasons:

1. To work around problems in the CPU architecture.
2. To speed up the guest OS run.

The first reason is fully discussed in Section 4.1.5 for x86 architecture. Generally, the x86 architecture contains certain operations that cannot be easily virtualized (e.g. storing interrupt description table), because they do not trigger interrupt when used in a bad context. It means that the problematic instruction (e.g. SIDT) must be replaced with a call to a *hypervisor* (this call is named *hypercall*). The hypervisor is another term for VMM.

One of the interesting features that allows VMM to achieve better speed is hardware resource allocation and dedication for one particular guest operating system. It means that VMM allocates a hw device to one guest OS and lets the guest to communicate straight with the hardware (so the guest is the one who implements the device driver). This is a violation of *"guest must not circumvent VMM"* statement and it is the price for better I/O performance.

Paravirtualization approach is used for example in projects: Xen[45], User Mode Linux[6], IBM POWER[51].

4.1.4 OS level virtualization

This type of virtualization does not present completely new execution environment for a process. Usually it only adds some new barriers and checks to an existing kernel. Resources segmentation is done on more levels than in usual operating system. It means that resources (e.g. processes, files) are grouped not only by users but also by the guest system[1] . The dotted line in the figure 4.2 shows the grouping level characteristic for OS level virtualization.

[1]The guest systems have many names depending on the implementation: Zone, jail, VPS.

Main representants of OS level virtualization are Linux Virtual Server[49], OpenVZ[27], Solaris Zones[24] or FreeBSD[50] jails.

4.1.5 Limits of VMM on i386 architecture

4.1.5.1 CPU requirements for VMM

This section summarizes the limatation in creating VMM for i386 (a.k.a. x86) architecture which would achieve native performance for running hosts. It summarizes the requirements for VMM defined by Goldberg in [31] and analysis of Pentium instruction set done by J. S. Robin in [36] and [35].

The key to natural performance is that statisticaly significant part of all instructions are executed straight on the processor. The remaining instructions have to emulated by VMM. If the portion of emulated instructions is 0% the VMM will be called *true VMM* otherwise it will be called *hybrid VMM*. If the emulated portion is 100% the VMM becomes an Emulator.

The main problem for VMM is the need to execute *sensitive* instructions. The instruction is sensitive if and only if:

- it reads or writes the state of the machine or virtual machine (VM)
- it reads or writes sensitive registers, memory locations, system clock, interrupt tables etc.
- it reads or changes memory management tables and registers
- it is an I/O instruction

If the sensitive instruction is to be executed, the VMM must be informed so it can emulate the behaviour of the instruction and limit its consequences only to current VM. So the CPU must be able to trap on such instruction if the VMM wants so.

This general requirement is satisfied on i386 architecture by 2 features:

- rings of execution
- privileged and non privileged instructions

Ring of execution or CPL[2] is a mode of CPU. Some CPL value is associated to every running task. There are four rings 0-3 and only task on ring 0 is allowed to execute privileged instructions. If the task A is running on CPL other than 0 and wants to do something involving privileged instruction (e.g. reading from disk) it has to ask other task B running on ring 0 to do it on A's behalf.

This is how typical operating system with userland processes work. The OS runs on CPL 0 and userland processes run on CPL 3. If we want to put VMM under the running OS and give VMM the absolute control over the machine, we have to put VMM on ring 0 and move OS away from ring 0. This brings the problem with instructions used in OS's code that assume they are executed on CPL 0.

So the instruction set of i386 architecture can be divided into 4 parts:

[2]The official acronym means Current Privilege Level

1. Ordinary instructions, not privileged, not sensitive for VMM. Such instructions pose no problem for virtualization.
2. Privileged instructions not sensitive for VMM. These instructions trap when executed on other CPL than 0. Typically VMM handles this trap without a problem but it is a performance loss.
3. Sensitive instructions which are not privileged. These instructions are a big problem, because they can circumvent VMM by not triggering a trap. This way the task running in VM can affect other VMs or the VMM itself. In Section 4.1.5.2 we will present an example of such instruction on i386.
4. Privileged instructions which are also sensitive for VMM. These instructions can affect VMM but they can be caught by the VMM and their influence is limited only to current VM.

4.1.5.2 Bad instructions on i386

There are many instructions that prevent i386 from being virtualized easily. We will mention only one as an example. The complete analysis can be found in [36].

Generally the problematic instructions are those which read the status of the machine. The typical instruction of i386 that falls to the part 3 mentioned in previous section is **SIDT**. This instruction stores content of IDT[3] to a general purpose register or memory location. It does not trigger interrupt when executed with privilege ring >0 so the VMM cannot fake the result according to particular VM and even non privileged ring gets the IDT of the physical CPU and not the virtual one.

4.1.5.3 Workarounds

One solution to presented problems is paravirtualization described in 4.1.3. The guest OS is patched so it does not use the *bad instructions* and reads the requested information in another way[4].

The other solution which does not require patching is **binary translation** (BT). The basic idea of BT is that VMM inspects the code which is to be executed by guest OS and looks for any bad instructions. If a bad instruction is spotted the VMM replaces it with some harmless instruction which traps and the VMM can simulate the instruction in the appropriate trap handler. This approach can bring significant performance loss and with caveats like self-modifying code the proper implementation of binary translation can be very difficult and tricky.

4.1.6 Support for virtualization in hardware

As was shown in Section 4.1.5 i386 architecture lacks certain features to be *VMM ready*. Both industry leaders (Intel and AMD) have their own solution which enhances i386 with virtualization features. This section presents both solutions and their comparison.

[3] Interrupt Descriptor Table is a register that contains the address of interrupt vector table.
[4] Usually by calling a VMM interface.

4.1.6.1 VT – Vanderpool

At the beginning of 2006 Intel started selling new CPUs with so called VT enhancement. VT stands for Virtualization Technology and is also known under its code name Vanderpool. VT is the technology that helps developers to come with VMMs that are more secure and have better performance. As stated in [46] "*a central design goal for Intel Virtualization Technology is to eliminate the need for CPU paravirtualization and binary translation techniques*". The following paragraphs describe technical details how typical i386 vs. VMM problems are solved by Intel in their VT.

The main changes that VT presents are:

1. If VM support is on, CPU always works in one of two new modes.
2. VMM can run outside any privilege ring (so the guest OS is not forced to move from CPL 0).
3. CPU supports fast rescheduling of guest operating systems.
4. CPU can trigger VMM's routine when some important event occurs (the triggers are widely configurable)

We will discuss these points a bit deeper now. The figure 4.3 (taken from [5]) shows the startup of VM aware system and basic interaction between VMM and guest systems.

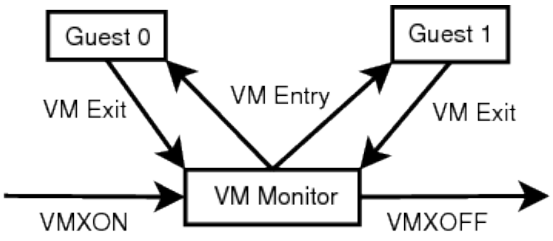


Figure 4.3: Intel VT basic schema

The virtualization support must be switched on on the CPU with the VMXON instruction. After this initialization, the CPU is in **VMX root operation** mode. It is a super privileged mode in which a typical VMM runs. In this mode VMM can allocate a memory for descriptor of a guest operating system. The descriptor is called **VMCS**. It can also turn off the virtualization support with the VMXOFF instruction. But the main action that VMM can do is so called **VM Entry**. It switches CPU to VMX non root operation mode and lets guest OS run. What guest OS runs depends on current VMCS (current VMCS is set by VMPTLRD instruction). The opposite action to VM Entry is **VM Exit**. When VM Exit happens depends heavily on settings in VMCS.

VMCS works as a storage for guest OS status data when the VM Exit happens. It also defines the conditions under which the CPU performs VM Exit. For example VMM can configure CPU to perform VM Exit every time when guest tries to access specific I/O port.

The VT defines and implements many other enhancements that make VMM programming easier (e.g. VMM can limit content of CR3 register and avoid VM Exits just for CR3 validity checks). Whole analysis and description is beyond the scope of this thesis, more details can be found in [5].

4.1.6.2 Pacifica

AMD's response to Intel's Vanderpool technology is their own solution for virtualization with official name Secure Virtual Machine (SVM) and codename Pacifica.

The basic approach of AMD is very similar to the Intel's one and we can say that Pacifica is a superset of Vanderpool in functionality. Both vendors use different naming for entities (see table 4.2) and unfortunately they also use different opcodes for intructions that have similar meaning (e.g. Intel's VMCALL vs. AMD's VMMCALL).

Intel	AMD	meaning
VT	SVM	Name of the virtualization technology.
VM Exit & VM Entry	World switch	The switch between host and guest.
VMX non root mode	Guest mode	The mode in which guest OS runs. Some instructions are limited.
VMCS	VMCB	Physical memory area which defines guest OS behaviour.
execute VMXON instruction	set EFER.SVME bit to 1	How the VM features are turned on.

Table 4.2: Intel vs. AMD terminology

It means that these VM technologies are incompatible although they use the same approach (new CPU mode, configuratibility of traps etc.).

4.1.6.3 Comparison

Both solutions present a major step in virtualization support for i386 architecture. Currently Pacifica contains more advanced features. Both companies are preparing next generation of virtualization features for their future processors so it cannot be said who is going to win "the virtualization contest".

4.2 Available solutions

In this section we describe various approaches to virtualization and resource separation. The list of projects is not complete, there are other important projects implementing virtualization[5] .

In Linux with standard kernel all processes are separated and they can interact only via defined interfaces (e.g unix sockets, IPC tools etc.). Information about running processes is usually accessible by all processes (and users) on the system.

[5]KVM and VirtualBox are two big projects that are not mentioned in this thesis. VirtualBox is very similar to VMware and KVM is not yet production ready.

In general we can say that there are only 2 levels of management in UNIX-like operating systems:

- user management – listing other's processes, administration for user's own processes
- root management – all administration ranging from process management to hardware resource management[6]

The following projects insert another layer between these two. They limit root privileges to some predefined area (disk area, process group). Or they virtualize hardware resources completely and give a user his own "computer".

4.2.1 chroot() system call

4.2.1.1 Description

Chroot is a representant of the first group. It simply sets root directory (commonly reffered to as one character: **'/'**) to another directory and executes given binary in context of this new root. Typically this binary is some shell which gives user (or application) completely different execution environment.

According to man page, *chroot()* syscall is a part of standard specifications SVr4, SVID, 4.4BSD and X/OPEN. In Linux, *chroot()* is implemented in quite a straighforward way. Every task keeps its root directory and this is changed in *sys_chroot()* function [7]. There are many ways how to interact with "outer world" from chrooted environment. There is no limitation for accessing information about other processes, devices or other resources.

Chroot is not a good solution for separation and it is mentioned here just for the sake of completeness. In Appendix A there is a detailed description of various methods of breaking out of chroot.

4.2.1.2 grsecurity enhancements for chroot

There has been some effort to harden chroot facility and one of the most complete solutions is contained in grsecurity project[12]. Grsecurity imposes following restrictions on chrooted process:

- chrooted process cannot call *chroot()* again
- mounting is forbidden in chroot
- the root of the process trying to call *shmat()* is compared to root of the process that has created the shared memory, if they differ the call fails
- calling *ptrace()* is forbidden for processes in chroot

[6] In Linux there is the possibility to restrict root by setting capabilities but it is a bit tricky and not used by wide community.

[7] See http://lxr.linux.no/source/fs/open.c for the function definition.

- file descriptor is validated to file and *fchdir()* fails if the file is out of chroot
- *pivot_root()* syscall is forbidden for chrooted process
- adding SUID bit to a binary is prohibited when in chroot
- kernel system parameters cannot be changed neither via *_sysctl()* nor via procfs
- chrooted process cannot connect to unix socket belonging to process outside of chroot

Additional list of potentialy dangerous actions for chroot restricted by grsecurity can be found in [13].

4.2.2 BSD jail

The first solution to many chroot problems came in FreeBSD operating system presented in [21]. It is called jail and consists of two system calls *jail(2)*, *jail_attach(2)* and utilities *jail(8), jexec(8), jls(8)*. Generally, we can say that jail is a chroot remade with regards to:

- security
- delegation

All the problems mentioned in Section 4.2.1 are solved in jails, because jails are designed with better security approaches. Delegation means that multiple administrators can be given one jail with preinstalled environment (resembling a real system) and each administrator can thus administer one and only application (good example can be BIND).

FreeBSD jail is a good solution comparing to chroot but it is also limited for our purposes:

- all jails share the same version of kernel
- jail is easily spotted (with no patches the proc filesystem reveals that by special flag for each process and lack of init process and kernel daemons)
- processes across jails can interfere with each other (defaul scheduler is not jail aware, but there are patches for this problem [8])
- security configuration of jails (via sysctl) is the same for all jails, they cannot be set per jail (e.g. access to IPC primitives is permitted for all jails or no jail)

Now let's present a brief example of building a jail. We start from the point where complete OS is installed:

- install minimal system to a directory
- setup jail with dedicated IP address

The jail can serve as a DNS server so we name it `dns_fbsd`, but we do not describe the installation and the configuration of BIND.

In Listing 1 and Listing 2 we have presented a very simple way how to setup jail. However there are other security aspects that must be considered and configured. These are for example devfs limitation (with devs rules).

Listing 1 Installing FreeBSD system to a directory.

```
# first download sources of FreeBSD and install it to /usr/src
$ mkdir -p /tmp/install
$ cd /tmp/install
$ wget -nd -r --no-parent \
ftp://ftp.cz.freebsd.org/pub/FreeBSD/releases/i386/6.0-RELEASE/src/
$ sh install.sh all
# now build the sources
$ cd /usr/src/
$ make buildworld
```

Listing 2 Building and starting a jail

```
# install built sources to a directory
$ mkdir /bind_jail
$ cd /usr/src
$ make installworld DESTDIR=/bind_jail
# configure and start jail
$ mount -t procfs proc /bind_jail/proc
$ mount_devfs devfs /bind_jail/dev/
$ jail /bind_jail dns_fbsd.elf.farm.particle.cz \
172.16.89.103 /bin/sh
```

4.2.3 Solaris zones

Another UNIX-like operating system with integrated solution for virtualization is Solaris. The solution in Solaris is called **zones** and it consists of system calls (*zone_create, zone_enter, zone_getattr, zone_list, zone_lookup, zone_shutdown, zone_destroy*) and userland utlities (*zoneadm, zonecfg, zonename*).

Solaris zones are quite similar to FreeBSD jails. Only privileged user is allowed to do zone administration tasks, such as creating new zone, deleting existing one or adding/removing resources attached to a zone. Even the root must do such tasks from one privileged zone called **global** zone.

Solaris zones provide more configuration options than FreeBSD jails. Zone can be limited in I/O operations, CPU occupation, memory ocuppation etc. The reason for this is the very good resource management in Solaris (for details see Resource Management chapter in [42]). Almost all resources can be limited on per zone basis.

The second main advantage comparing to jails is the user interface for managing zones. Solaris installation system supports zones so the administrator does not have to install new instance in a (chrooted) directory and then set it up. Zone is first configured and then installed (installation process is very space conscious) via standard Solaris installation mechanism. Then new zone is booted and a console attached to it. Administrator can then setup for example SSH server inside the zone and connect to it from outside world.

As already mentioned fresh zone installation occupies only very small disk space (about 100 MiB). This fact is achieved usually via loopback mount filesystem *lofs(7FS)*. Many directories (e.g. /opt) can be loopback-mounted to the global zone.

Let us summarize the pros of zones:

- regular user can be set to be the administrator of a specific zone

- many configuration options such as
  - resource management (zone which uses resource management is called a container)
  - zone can have its complete network configuration
  - non global zone can share directories with global zone (option inherit-pkg-dir)
  - zone can be limited in its bandwitdh

And some cons:

- IPFilter cannot be applied on communication between zones
- non-global zone cannot work as an NFS server
- swap utilization cannot be limited on per zone basis

Finally let us present an example of simple zone initialization (to show how simple it is). In Listing 3 there is a basic setup of zone with a dedicated IP address.

Listing 3 Zone definition

```
$ mkdir /zone1
$ chmod 700 /zone1
$ zonecfg -z zone1
# since now we are in zone configuration utility so
# the prompt is different from the one in our shell
zonecfg:zone1> create
zonecfg:zone1> set zonepath=/zone1
zonecfg:zone1> set autoboot=false
zonecfg:zone1> add net
zonecfg:zone1:net> set address=172.16.89.99/24
zonecfg:zone1:net> set physical=pcn0
zonecfg:zone1:net> end
zonecfg:zone1> verify
zonecfg:zone1> commit
```

When the zone is configured it should be installed and booted. We can also first inspect our configuration. All these steps are in Listing 4.

4.2.4 VServer

Linux VServer project [32] brings OS level virtualization to Linux kernel. The units of isolation are called *contexts*. The project consists of two key parts:

- patch adding support for contexts to Linux kernel
- user space utilities

The main features of the project are[11]:

- Filesystem isolation for every context (based on chroot(2) and file tagging).

Listing 4 Zone installation and boot

```
$ zonecfg -z zone1 info
zonepath: /zone1
autoboot: false
pool:
inherit-pkg-dir:
        dir: /lib
inherit-pkg-dir:
        dir: /platform
inherit-pkg-dir:
        dir: /sbin
inherit-pkg-dir:
        dir: /usr
net:
        address: 172.16.89.99
        physical: pcn0
# some directories are automatically inherited so the new zone
# occupies as small space as possible
$ zoneadm -z zone1 install
$ zoneadm -z zone1 boot
$ zlogin -C zone1
# now we are attached to new zone and we have to configure
# newly created solaris installation (hostname, security etc.)
```

- Process isolation – Every process can see only processes created in the same context (except context **1** that can see everything).
- Network isolation – Each context has its own IP address and hostname which cannot be changed from within the context.
- IPC isolation – All IPC primitives are assigned (and limited) to one context.
- Root limitation – Only root in context 1 can access manipulate devices (e.g. mount disks, reconfigure network interfaces).
- Ported (and tested) on most architectures supported by Linux kernel: alpha, ia32/ia64/xbox, x86_64, mips/mips64, hppa/hppa64, ppc/ppc64, sparc/sparc64, s390, uml
- Large community with support (irc channel, mailing list, wiki page with many howtos).

This project uses as much as possible from existing kernel abilities to separate contexts and adds only necessary patches to Linux kernel. For example the patch for kernel version 2.6.7 adds only 6836 lines.

The development of VServer has been forked by Positive Software Corporation in 2003. The new project is called FreeVPS.

4.2.5 OpenVZ

OpenVZ or sometimes open Virtuozzo is another OS level virtualization for Linux kernel which calls one unit of isolation *Virtual Private Server* (VPS). It is an open source version of project Virtuozzo[28] by Parallels (formerly SWsoft).

This project is very similar to VServer in implementation and also in usage. The main differences of these two projects are how comfortable their setup is, how they manage resources and how they administer the filesystem. The following list shows the advantages of OpenVZ over VServer:

- Virtual network devices – this feature allows administrator to set VPS's networking independent on the host. For example, every VPS can have its own iptables rules set within the VPS (VServer uses only one network device with various IP addresses attached to particular context).
- User beancounters – More kernel resources are limitable by the administrator, for example network buffers.
- Template system – A set of preinstalled Linux distributions with supporting tools for easy deployment of a VPS.
- OpenVZ is supported by commercial company and oriented on RHEL[8] so it is more appropriate for enterprise customers.

There are also some disdvantages of OpenVZ comparing to VServer:

- Support for less hardware architectures.
- Smaller user community.

4.2.6 QEMU

QEMU, a *"Fast and Portable Dynamic Translator"* [1] is an open source project with two aims:

- execute Linux binaries compiled for one architecture on a different architecture
- run one operating system inside another

For our purposes we will focus only on the second aim which is also called "full system emulation". In this mode QEMU emulates not only CPU but also other peripherals like a network card, USB, a sound card or a graphics card.

The guest OS is usually installed in a disk image created by *qemu-img* utility. QEMU can emulate booting from cdrom so typical installation procedure can be performed. After succesful installation the guest can be rebooted into the installed OS.

The technique used for emulation is binary translation, so QEMU is much slower than OS level virtualization projects mentioned above. On the other hand it is capable to run complete and unpatched guest OS independent on the host OS.

QEMU comes with an accelerator module called KQEMU. It is a kernel module that must be inserted to the host OS. It speeds up emulation of OS compiled on i386 and running on i386 architecture. The speed is about 10 times higher then with pure QEMU.

[8] Red Hat Enterprise Linux

4.2.7 VMware

VMware is a company providing many products focused on virtualization. Its main products are: VMware Workstation, VMware Player and VMware ESX server.

VMware Workstation is a virtual machine monitor very similar to QEMU with the accelerator module (KQEMU). It emulates basic PC hardware (CPU, harddisk, sound card, graphics card, USB, network card). And lets a user to run an unmodified guest OS. Its performance is similar to QEMU+KQEMU.

Advantages of VMware over QEMU:

- More user friendly – VMware has a nice GUI for setting up new virtual machine
- VMware tools – guest OS specific device drivers that offers special features like: better screen resolution for graphics, file sharing between guest and host, copy'n'paste between host and guest etc. Please note that these tools must be ported and installed in guest OS.

Advantages of QEMU over VMware:

- Easier installation – VMware must be reconfigured for every new kernel[9]

Until recent times the VMware was strictly commercial software. Because of big competition on the market of virtualization solutions the company decided to provide a free VMware Player. As its name suggests the program can be used only for running prepared virtual machines. It cannot create new ones. But this was only a small problem because format of virtual machine files has been decoded and free utilities for VM assembly are now available freely.

VMware ESX server is a completely different architecture rather similar to Xen (see next section). It is a hypervisor that puts only a thin but highly secure layer between guest OS and hardware. It is based on patched Linux kernel and oriented on enterprise customers so most known features are:

- support for SAN storage
- VMotion – the ability to migrate life (running) virtual machines between two ESX servers
- VirtualCenter – monitoring and management tool for multiple ESX servers
- API for automatic guest startup and deployment

The main disadvantage (compared to Workstation) is limited hardware support.

4.2.8 Xen

The Xen is an open source lightweight hypervisor. It is using paravirtualization to run various operating systems as its guests. Xen is built on simplified Linux kernel and presents idealized hardware to its guests. This means that mainly device drivers have to be adapted in the guest operating system.

[9]The KQEMU has to be reconfigured for new kernel too. So this advantage is not so significant.

But Xen is also the first project that supports Intel's VT (see Section 4.1.6.1 for more details) and AMD's Pacifica (see Section 4.1.6.2). Those are CPU features for virtualization. So even an OS that cannot be modified to run in paravirtualized environment (e.g. Microsoft Windows) can run in Xen. The virtual machines using this *hardware virtualization* are called HVM.

A unique feature of Xen is its virtualized I/O subsystem. As was already said the drivers in guests are replaced by very simple drivers that communicate with the idealized hardware of the hypervisor. The hypervisor then delegates communication from guests drivers to one driver (per device) that is a part of the domain 0.[10] This allows Xen to recover from driver failure without rebooting.

Main advantages of Xen are:

- good performance (low overhead)
- support for virtualization hardware
- support for live migration of guests between two physical machines running Xen
- good granularity in resource management

One of the main disadvantages of Xen is a bad support for Windows on hardware without virtualization support.

4.2.9 Performance tests

We have done some benchmarks to compare these projects from the performance point of view. This comparison is not very fair because as was already mentioned the projects utilize different approaches to virtualization, they have very different feature sets and are targeting different user's needs.

However these tests still have a value in showing what solution is out of a question for implementing honeynet and what approach is acceptable. All tests were performed on a Compaq Evo N115 notebook with parameters summarized in table 4.3:

CPU vendor	AMD
CPU model	Duron
CPU cache	64 kB
Bogomips	997.67
Total memory	239872 kB

Table 4.3: Test machine characteristics according to /proc

[10] Domain 0 is a virtual machine that has rights to manipulate the hypervisor. It is also sometimes called the driver domain.

4.2.9.1 ubench

Ubench is a very simple benchmark. As mentioned at the homepage[30]: “Ubench is executing rather senseless mathematical integer and floating-point calculations for 3 mins concurrently using several processes, and the result is Ubench CPU benchmark.”. Results of the performed test are in the figure 4.4. The result numbers (y axis) express the number of operations done by ubench during the three minutes.

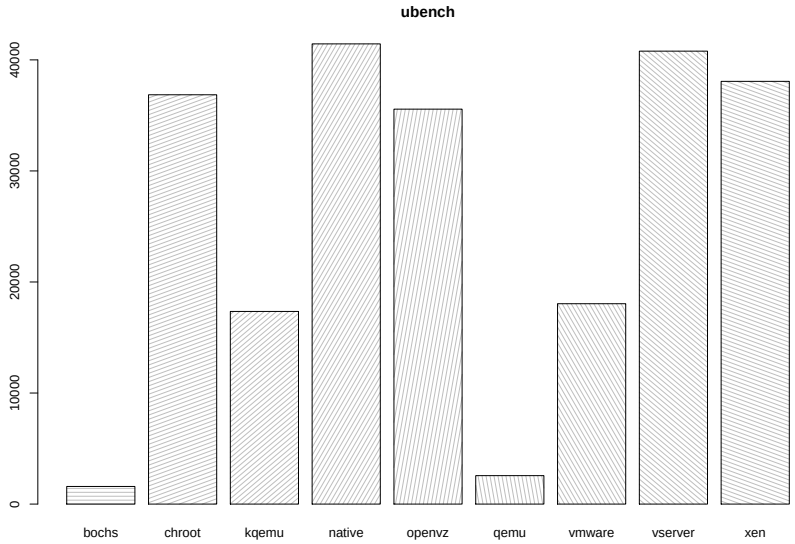


Figure 4.4: Ubench graph

4.2.9.2 nbench

Nbench is a Linux port of BYTE Magazine’s BYTEmark benchmark program. Its benchmarking routines focus on system’s CPU, FPU and memory abilities. All of the included tests are checked for statistical consistenty. It means that if one run of the test varies too much from other runs then the run is discarded from results and is performedagain. It can be performed even on production system because the results are not affected by resource peaks of other applications.

The included tests are (from nbench’s README):

Numeric sort Sorts an array of 32-bit integers. Exercise non-sequential performance of cache (or memory if cache is less than 8K).

String sort Sorts an array of strings of arbitrary length. Tests memory-move performance.

Bitfield Executes a variety of bit manipulation functions. Tests bit alteration and moving in place.

Emulated floating-point A small software floating-point package.

Fourier coefficients A numerical analysis routine for calculating series approximations of waveforms. Tests mainly FPU performance, it is not independent on memory performance.

Assignment algorithm A well-known task allocation algorithm. The test moves through large integer arrays in both row-wise and column-wise fashion.

Huffman compression A well-known text and graphics compression algorithm. A combination of byte operations, bit twiddling, and overall integer manipulation.

IDEA encryption Block cipher algorithm.

Neural Net A small but functional back-propagation network simulator. Small-array floating-point test heavily dependent on the exponential function

LU Decomposition A robust algorithm for solving linear equations.

The tests were performed with default options and compiled with gcc 3.2.3. The results can be seen in the figure 4.5. The shown results are comparison to results achieved on Pentium 90 with 256 KB L2-cache with nbench compiled with Watcom compiler 10.0.

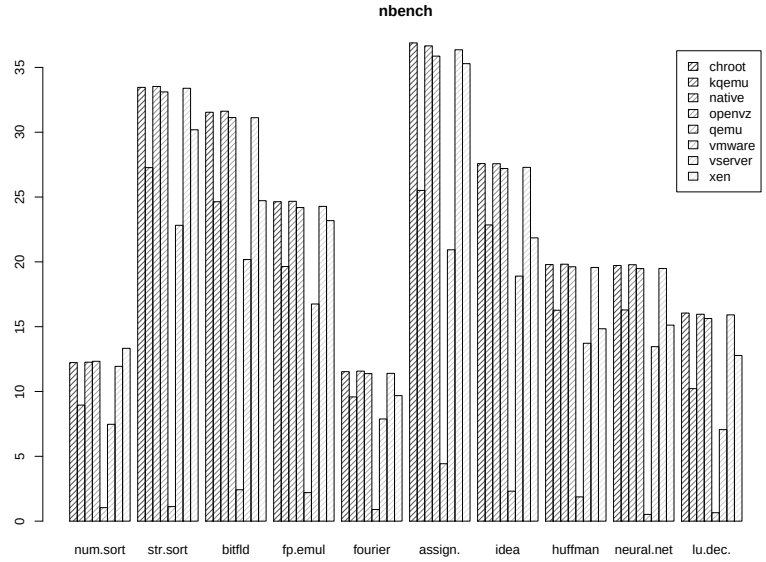


Figure 4.5: Nbench graph

4.2.9.3 FreeBench

The FreeBench[9] is “a totally free benchmark for computer systems”. It is developed with concerns about openness, platform independence and good balance. The included tests are:

Analyzer Tool for analyzing memory access traces for data dependences.

FourInARow Program that plays a game of "four in a row" against itself. It uses a min-max method with alpha-beta pruning as a search algorithm.

Mason This program solves a puzzle. It uses only integer arithmetics, and has a very small data set. This is a pure clock frequency test.

pCompress A file compressor using a three stage approach. Burrows Wheeler block-sorting, run length encoding and Arithmetic coding. The program is quite memory intensive.

PiFFT The program uses a huge FFT to calculate many decimal places of PI. Stresses FPU is very memory intensive.

DistRay A small ray tracer using random ray distribution to achieve anti- aliasing and soft shadows. FPU intensive while not memory intensive.

NeuralNet A neural network doing character recognition. Very memory intensive.

The result of the tests can be seen in the figure 4.6. This time the results express comparison to Sun Ultra10, with a 333MHz UltraSPARCIIi and 2MB L2 cache (this means score 1.0).

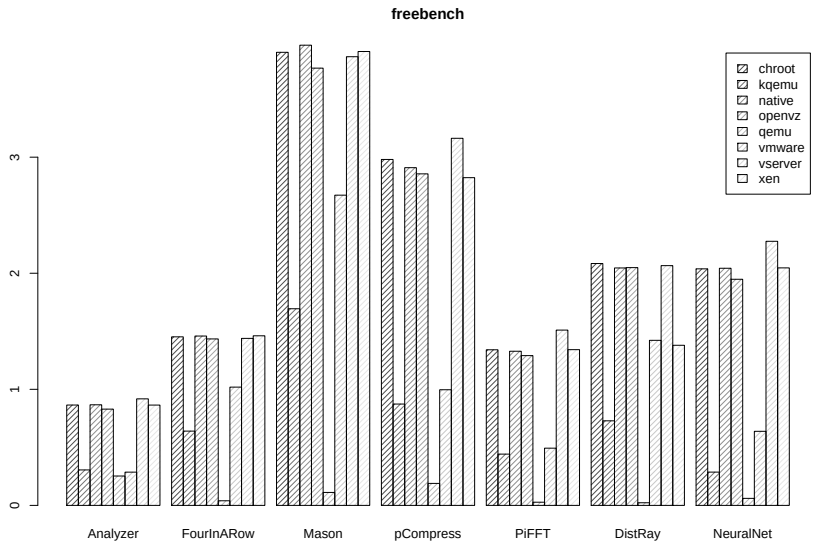


Figure 4.6: FreeBench graph

4.2.10 Performance conclusion

There are some anomalies in the tests results. There is no reason for chroot to be slower than native or VServer. But the results are sufficient for the overview of the

virtualization architectures and their performance. The test results let us compare the architectures with each other.

As expected the best performance is achieved by projects in category *OS level virtualization* which is on the other hand the category with the poorest featureset. The other side of performance table—emulation—can be considered a complete computer (even with its own BIOS). Although this is fantastic for pretending real system the performance is a stopper here.

The paravartualization is the answer to our question what project to use. It offers satisfactory performance, quite full featureset and its future is the most promising. That is why we have chosen Xen, whose features were presented in 4.2.8. The truth is that the paravirtualized environment can be simply identified[11]. It does not pose a problem as we will explain in the Section 4.3.

There are other projects implementing some kind of virtualization for various platforms. However we have mentioned the most used representants of the virtualization approaches. The future of virtualized computing is bright because the main players at CPU market (Intel and AMD) both have their solution for hardware supporting virtualization.

4.3 Detecting virtualized environment

When the virtual machines started being popular they were used mainly for testing, development of operating systems, honeypot deployment etc. The administrators of production systems were sceptical about the virtualization. For attackers this fact meant that the virtual machines were not useful for them and sometimes even dangerous for their malicious intentions.

As was described in this chapter, the virtualization is never perfect. Mainly for performance reasons and for simplicity of the implementation the virtual machine does not actively hide the fact it is virtual. So the black hat community developed several countermeasures and algorithms that can test whether the machine they operate on is physical or virtual.

One such technique exploits the relocation of cpu tables (LDT, GDT, IDT) which is done in many VMMs[34]. Great article by Thorsten Holz and Frederic Raynal [17] describes other approaches to revealing virtual machines. The authors analyze several VMMs and show the way they differ from a real system.

However in these days the virtualized environment is widely popular in production systems too. The VMMs are stable enough, have good performance and they bring additional advantages for the administrators and users. As an example of big deployment of virtualization we can name project Magrathea[20] – virtual machines manager and scheduler used and developed by Czech centre for high performance computing *Metacentrum.*[12]

[11] Xen for example can be identified by the existence of /proc/xen subdirectory.
[12] http://meta.cesnet.cz/

Chapter 5

Implementation

We start this chapter with the description of our virtual honeynet. Then we present user simulation techniques and our model for simulating users. The main part of the chapter is the description of *VAccess library* and its usage. It is a library that allows a programmer to inspect and manipulate guest virtual machines (honeynet members) and so effectively build a virtual honeynet with monitoring capabilities. At the end of the chapter we show a simple application using this library to inspect the running honeypots.

5.1 Virtual honeynet (cluster) deployment

The honeynet we will build is similar to a typical local computer network oriented on cluster scientific computations.

It means there is one machine that works as a gate to the Internet for all other machines and also as a *head node*. The head node function means that the users can connect (using SSH) to this machine and work on the local network through this connection. User's typical work consists of preparing a computational job and sending it into a batch system. The job is usually a shell script that downloads some data and process them with a special software.[1]

The job system is a program that distributes the jobs to other machines in the local network. These machines are simple Linux boxes with no special function. Their main task is only to retrieve a job, execute it and return results. These nodes are called *worker nodes.* Our head node and all the worker nodes will run on one physical machine with Xen as Virtual Machine Monitor.

This section describes installation of the physical host, installation of the virtualized guests, configuration of a one guest to be the head node, configuration of two guests to be the worker nodes and configuration of the network in Xen so the guests form a virtual network.

[1] The special software is beyond the scope of this thesis. It is for example Matlab, Wien2k, Fluka etc.

5.1.1 Host installation

The first thing to install and configure is the dom0[2] which means to install some Linux distribution with Xen-enabled kernel. We have chosen Scientific Linux[3] version 5.2. The only steps different from a default installation were adding the xen and kernel-xen packages. When the installation is completed we boot the xen kernel and continue in setting up the guests.

5.1.2 Guest installation and Xen configuration

Xen is able to load and start a guest in several ways. The most common is having a guest partitions in files (one file per partition) or on LVM[4] volumes. We have chosen the former approach, but both of them are equally good.

We have created two partitions (files) for each guest. One for the root partition and one for swap. We have mount the root partition via a loopback device to the directory `/mnt`. Then we install the operating system to the directory `/mnt`. Many Linux distributions have their way to install the operating system into a directory. The details of guest installation are described in appendix B.

The Linux distribution that we have chosen for guests is Scientific Linux version 4.5 because it is widely used in the community we want to mimic by our honeynet.

The xen configuration for head node is in the figure 5.1 and the xen configuration for a worker node in the figure 5.2.

```
kernel = "/boot/vmlinuz-2.6.21-7.fc7xen"
ramdisk = "/boot/initrd-2.6.21-7.fc7xen_koubat.img"
memory = 256
root = "/dev/sda1"
extra = "console=xvc0 3"
disk = ['tap:aio:/home/xen_imgs/sl4ts_root.img,sda1,w',
'tap:aio:/home/xen_imgs/sl4ts_swap.img,sda2,w']
name = "sl4ts"
vif = ['vifname=sl4ts_eth0,mac=00:16:3E:40:A0:00,bridge=xenbr',
'vifname=sl4ts_eth1,mac=00:16:3E:40:A0:01,bridge=localbr']
```

Figure 5.1: Xen configuration for the head node.

```
kernel = "/boot/vmlinuz-2.6.21-7.fc7xen"
ramdisk = "/boot/initrd-2.6.21-7.fc7xen_koubat.img"
memory = 256
root = "/dev/sda1"
extra = "console=xvc0"
disk = ['tap:aio:/home/xen_imgs/sl4tc1_root.img,sda1,w',
'tap:aio:/home/xen_imgs/sl4tc1_swap.img,sda2,w']
name = "sl4tc1"
vif = ['vifname=sl4tc1_eth0,mac=00:16:3E:40:A1:00,bridge=localbr']
```

Figure 5.2: Xen configuration for a worker node.

[2]We use the terms *dom0* and *host* interchangeably. Similar to that, we also use *domU* and *guest* interchangeably.

[3]Scientific Linux (or SL) is a Linux distribution produced by Fermilab and CERN. It is based on Red Hat Enterprise Linux. It can be found at http://www.scientificlinux.org/

[4]Logical Volume Manager – http://tldp.org/HOWTO/LVM-HOWTO/

5.1.3 Network topology

The topology of the virtual network that we use is shown in the figure 5.3.

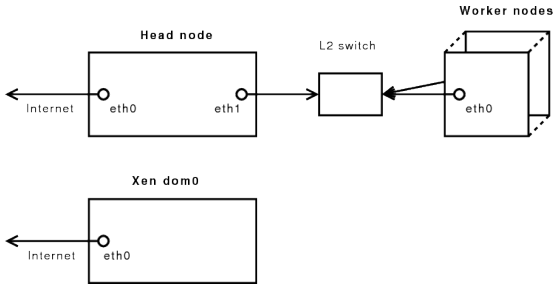


Figure 5.3: Virtual network topology

The configuration with network bridges and "invisible" virtual interfaces is shown in the figure 5.4. There are two network bridges. Each of them exists for a different reason.

Xenbr0 connects the physical interface of the dom0 (the interface is called peth0) with a virtual interface in dom0 belonging to head node[5] (hn_eth0). This allows the interface eth0 in the head node to have a public IP address and to be accessible from the Internet.

Localbr connects virtual interface in dom0 belonging to the head node (hn_eth1) and all virtual interfaces in dom0 belonging to worker nodes (wnX_eth0 where X is the number of a worker node). This bridge represents (and implements) the local network and the L2 switch in figure 5.3. The head node works as a default gateway for the local network. The head node also performs NAT[6] so the worker nodes can access the Internet, even though they are in a private network.

The network setup is implemented in initialization script */etc/init.d/va_honey*, which is also enclosed on the thesis CD.

[5]There is a virtual interface in dom0 for every network interface in a virtual machine. They are connected with a virtual cable.

[6]Network Address Translation.

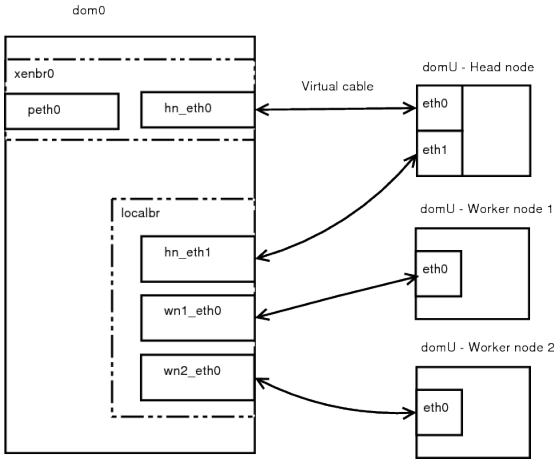


Figure 5.4: Network configuration

5.1.4 Head node configuration

The head node offers the following services:

- SSH server so the users can login.
- Batch system server called torque, which allows users to submit jobs and compute them on the worker nodes.
- Network services for the private network. The network is called *koubat*.
  - Routing services. It means turning on the IP forwarding and turning on the NAT.
  - DHCP server. We use dnsmasq[7] that gives hostnames and IP addresses according to its static configuration file.
  - DNS server. We use the same dnsmasq for DNS as well. It is configured to resolve according to */etc/hosts* file.

5.1.5 Worker nodes configuration

The configuration of a worker node is more simple. It is configured to obtain all network information via DHCP client. The main service, that each worker node provides, is computing the jobs. This is done by torque client. Its configuration contains only the name of the server in */var/spool/pbs/mom_priv/config*.

5.2 Simulation of a user

When a honeypot is compromised we can assume that the attacker gains root access.[8] Even without the root access an attacker can start doing malicious things on behalf

[7] http://www.thekelleys.org.uk/dnsmasq/

[8] There are many known exploits "in the wild" and skilled attackers usually know of a security vulnerability in a production software. We can mention vmsplice exploit as an example: http://www.milw0rm.com/exploits/5092

of an ordinary user. A completely empty machine would be very suspicious for the attacker. She would leave the machine then and we would have no interesting data to study.

If we want our honeypot to look like a real production service we need to simulate some activity on the honeynet. There are several production services that could be simulated, we have chosen the following for our target server:

- SSH server (to issue commands on the target server)
- torque server[9] (to simulate some real work on the cluster)

To simulate a user who is performing an SSH session, we need three main building blocks:

1. a model of a user behavior (what commands are used, how the commands are serialized etc.)
2. a model of a user typing habits (how quickly the user types, how many typing errors are normal for the user etc.)
3. a way how to issue the modelled commands (and keystrokes) and send them to the SSH session

The building block 1 is often studied in intrusion detection based on anomaly detection in user behavior [16, 38, 37, 4]. We have chosen to implement an approach very similar to the one specified in [4]. The details of the model are described in Section 5.2.1.

The building block 2 is rather complementary and we describe it in detail in Section 5.2.2.

The third building block is a glue to get everything together. It should start new SSH connection, choose commands to issue (apply the behavior model), send keystrokes via the SSH connection (apply typing model) and close the connection. We describe this glue in Section 5.2.3.

5.2.1 Behavior model

Our behavior model consists of commands, meta-commands, jobs and sessions:

- command – Typical command used in a shell session (e.g. ls, less, cat, vim, ssh, firefox).
- meta-command – Set of commands with the same function (e.g. "file content viewer" meta-command consists of commands less, cat and more).
- job – Sequence of meta-commands that perform certain task (such task can be for example creating a file and listing it). The sequence is not always strict. The meta-commands are executed in a random fashion and their order is described by a transition matrix. The first meta-command in job is selected from the job's initial vector. It is a vector of probabilities of each meta-command in the job.
- session – A set of independent jobs form a session. The jobs can be executed sequentially or in parallel.

[9] Torque is an open source implementation of Portable Batch System. Its homepage is at http://www.clusterresources.com/pages/products/torque-resource-manager.php

We use XML format for saving the behavior model. A python class implementing the behavior model is called *CommandModel* and it is stored in python module `sources/simulations/command_model.py` on the thesis CD.

This model however has one problem. It lacks support for return values and error strings. The model simulates the actions disregarding the success or failure of the last command. In the articles we mentioned[16, 37, 4], all the models ignore the return values of previous commands. Enhancing the model with some feedback functionality would increase its robustness and credibility. We leave this enhancement for future work.

5.2.2 Typing model

If we want to simulate an SSH session we cannot issue whole commands in one row. We have to send character by character so we can mimic a user typing on his keyboard.

Our model has two main components:

- Matrix representing delays between two characters.
- Overall (typing) error probability.

Typing model is implemented in python module `sources/simulations/typing_model.py` in class called *TypingModel*.

5.2.3 Application of the models

The main application that loads the models and simulates the SSH session is implemented in `sources/simulations/simulator.py`. It is a python script that performs several steps:

1. Connect to an SSH server
2. Apply the behavior model to get commands.
3. Apply the typing model on the generated commands and send the characters (a.k.a. "keystrokes") to the SSH connection.
4. Continue sending the characters as long as the connection is open. The connection is closed by a command (usually *logout*) generated by the behavior model.

The SSH connection and the communication is done via pexpect[41] library. The connection can be done from any machine out of the honeynet. Such connection mimics a real user connecting to a head node from his own computer to perform some work. We were launching the simulator from dom0 because it is the outer world for head node too.

5.2.4 Other applications and protocols

There are of course other applications that can run on our head node and that can communicate with simulated users. The head node can, for example, run a web server

and the users can browse the web pages served by this web server. Simulation of a user requesting pages in a semi-random manner is discussed in [38]. The probabilistic methods can be also used for other protocols like IMAP, POP3 or FTP. Complete and convincing implementation of simulation of these protocols is very hard, time consuming and beyond the scope of this thesis.

5.3 Monitoring the honeynet

In this section we present a library that allows a programmer to monitor our honeynet. We also make a simple example of the usage of the library. Finally we present a GUI application that uses the library.

5.3.1 VAccess library

The VAccess library allows introspection of a running guest. It is oriented on getting information from a Linux kernel. The code is based on XenAccess library[10], but enriched with functions and oriented only on paravirtualized Linux guests.

The functions offered by the VAccess library can be divided into two groups: low level and high level.

Main low level functions:

- va_get_guest_kernel_mem
- va_get_guest_user_mem
- va_symbol_address

Main high level functions:

- va_linux_get_all_tasks
- va_linux_get_task_by_pid
- va_linux_get_all_modules
- va_register_hook

va_init This function initializes one instance of the VAccess library. It expects domain id (number) that will be associated with this instance of VAccess library. If a user wants to use special features (like syscall hooking) it must be said during initialization with *flags* parameter.

The function returns success or failure. In case of success it also returns a VAccess handle that should be passed as a parameter to all other VAccess functions described below.

va_destroy Cleanup function that should be called for every VAccess instance.

[10] http://xenaccess.sf.net/

va_get_guest_kernel_mem This function returns a copy of a memory chunk from the kernel of given domain. It expects two parameters: the address and the size of the requested chunk.

va_get_guest_user_mem This function returns a copy of a memory chunk from a user space process running in the domain associated with current VAccess domain.

It expects three parameters (except the VAccess handle): the address of the requested chunk, the size of the chunk and the PID.

va_symbol_address Translates a symbol name to an address for a kernel running in the domain specified by a VAccess handle.

va_linux_get_all_tasks Retuns a list of all tasks (with their names and PIDs) that are currently running in the given domain.

va_linux_get_task_by_pid Returns info about a task with the given PID in the given domain.

va_linux_get_all_modules This function returns a list of all loadable kernel modules that are currently loaded in the given domain.

va_register_hook Registers a hook in the given domain. Supported hooks are for system calls *read()* and *write()*. This function allows the caller to wait for the system call and get the data that are processed by these system calls.

This feature is useful for getting data that go through these system calls. This way we can for example watch an SSH session (the commands used by an attacker on our honeypot) although the connection is encrypted.

5.3.2 VAccess configuration and logging

The VAccess library has got its own configuration file `/etc/vaccess.conf`. It contains global options for the library and also options specific for a domain.

The global options are:

- loglevel – The level of logging. Ranges from 0 (errors only) to 3 (very detailed log).
- logtype – The type of logging determines where the log messages shall go. There are four types of log targets: *file*, *stdout*, *syslog* and *none*.
- logfile – This option specifies the file where the messages shall go. The option is ignored when the log type is other than *file.*
- logtruncate – *true* or *false*. Tells the logger to truncate (or not) the file with logged messages. This option is valid only when the log type is *file.*

Listing 5 VAccess configuration example

```
{
    loglevel = "3";
    logtype = "file";
    logfile = "/var/log/vaccess.log";
    logtruncate = "false";
}
sl4ts {
    sysmap = "/boot/System.map-2.6.21-7.fc7xen";
}
sl4tc1 {
    sysmap = "/boot/System.map-2.6.21-7.fc7xen";
}
sl4tc2 {
    sysmap = "/boot/System.map-2.6.21-7.fc7xen";
}
```

There is only one domain specific option now:

- sysmap – Path to a `System.map` file. It must be the system map belonging to the kernel running inside the guest. This file is used for translating a symbol to its address in kernel space.

There is an example of `vaccess.conf` in listing 5.

The VAccess library writes messages into the system log before this configuration is read.

5.3.3 VAccess internals

VAccess library uses functions from *libxenctrl* (Xen control library). The functions of this library allow a programmer to map memory areas from the guest, manage event channels, virtual IRQ etc. All these functions are independent of the operating system running in the guest. It means that if we want to interpret the content of memory and give some meaning to the data we get via *libxenctrl* we need another layer. For Linux systems, VAccess works as this layer.

Kernel symbols addresses The VAccess library uses `System.map` files to obtain the virtual address of a symbol in guest kernel. The path to the file is specified in configuration file and it is domain specific.

Getting kernel-space memory The kernel memory in Linux on x86 is mapped in the last gigabyte of the virtual space. It means that all addresses higher than `0xC0000000` belong to kernel. The physical address can be obtained simply by subtracting the constant `0xC0000000` from the address. The physical address is than simply translated into *machine frame number* (internal Xen page number) by function *pfn2mfn()*. The obtained machine frame is mapped by a function from xenctrl library and copied to a temporary memory buffer. This is done for every page that programmer requests. Finally the temporary buffer is returned.

The library copies the data because it wants to give a programmer a safe API that does not allow him to perform an operation that would harm the guest operatin system.

Getting user-space memory Getting memory from userspace is a bit more complicated, because the mapping from virtual memory address to a physical address must be done via kernel internal page tables. More detailed description of this translation can be found in Mel Gorman's book [2].

When the VAccess is getting the requested memory it first get the page tables for given process from the running kernel. Then the translation is executed. The obtained physical address is then treated in the very same way as in the previous section.

Interpreting kernel data structures One of the problems in getting data from a kernel memory is that we do not know the layout of the memory. Especially data structures. Even if we know the address and size of a structure we do not know the offset of a structure member which is important for us. These offsets change between Linux kernel versions and they also depend on compilation options.

The vaccess library offers a convenient way to get these offsets automatically from the kernel build tree during the build of VAccess library. It means that if a programmer wants to use the library with his kernel, he has to rebuild the library.

The automatic mechanism is called *offset generator* and it is implemented in `sources/libvaccess/src/offset_generator`. The generator uses a data file `needed_offsets.dat` which contains the names of the structures and their members. This data file (during the build process) is used to generate a kernel module which is just compiled (not inserted in the running kernel). The object file of the kernel module is then used by the offset generator to create a C header file which contains constants in format `VA__STRUCT_<struct_name>__<offset_name>`. The file is then copied to VAccess build tree as a normal C header file. Its name is `sources/libvaccess/src/linux_offsets.h`.

Now the library is able to get the address of a kernel symbol, get the memory from guest and interpret right the offsets in kernel structures. It is then simple to obtain information like modules list or process list of the guest.

System call hooks The Xen allows a programmer to create an *event channel*. It is a communication channel that can be bound to a special events in a guest.

VAccess creates one event channel and connects it to a virtual IRQ[11] for debugging exceptions. It puts the breakpoint instruction on the right place (the end of the syscall we want to hook). The instruction replaced by the breakpoint is stored and put back when the hook is removed.[12]

When the hook is actually switched on (function *va_register_hook()*) the library forks a new process and returns a file descriptor of a pipe. This new process is the entity that processes the debugging events returned by Xen and feeds the data recieved from Xen to the pipe, so the programmer gets a stream of data going through the system call.

[11] Interrupt ReQuest

[12] In fact the stored instruction is put back for one CPU step every time the breakpoint is triggered.

The data we are talking about are in fact copied from kernel memory when the Xen pauses the domain because of debugging exception thrown by our breakpoint instruction.

Currently VAccess implements hooks for *read()* and *write()* system calls.

5.3.4 Example usage of the VAccess library

The simplest example of VAccess usage is listing all processes of a given virtual guest. We will present a situation where the attacker already got the root access and installed a rootkit (implemented as LKM) which hides specific processes from the procfs filesystem.[13] Then we will see that VAccess can show us that there is a hidden process in the guest system.

5.3.4.1 Example rootkit

There is an example rootkit in `sources/hide_proc` on the thesis CD. Its function is to hide processes which are running binary containing string 'HIDEME'. It logs all actions into syslog, so it is not a real rootkit, it just demonstrate the technique.

5.3.4.2 Running hidden process

When we have the hiding module, we can make some evil binary, compile it, name it HIDEME and run it. The root will still see the process HIDEME but when we insert the hiding module into kernel, the process magically disappears. The first and second column of listing 6 show the difference between output of command "ps -e -o pid,comm" in kernel without and with the hiding module.

The VAccess library has the ability to get all the processes as guest kernel sees them and its scheduler plans them. So even when they are hidden from procfs, the library is able to spot them (see the function *va_linux_get_all_tasks*). In the third and fourth column of listing 6 there is the comparison between two lists of processes as printed by `sources/libvaccess/examples/process_list.c` with and without the hiding module.[14]

5.4 VAmonitor – VAccess monitor

As an example what can be done with VAccess library we have developed an application VAmonitor. It is an application with a GUI built on Qt4 library.

The list of the features of the **VAmonitor** application:

- List the active domains.
- List the processes in the selected guest.

[13] Procfs is usually mounted in */proc* and contains mainly information about running processes. For more information see procfs(5).

[14] The last *ps* process shown in 1st and 2nd column is the process used in the guest for listing processes.

Listing 6 Comparison of process list in various situations (output similar to “ps -e -o pid= -o comm=”)

Inside guest without the hiding module.	Inside guest with the hiding module.	Access via VAccess w/o the hiding module.	Access via VAccess w/ the hiding module.
1 init	1 init	1 init	1 init
2 migration/0	2 migration/0	2 migration/0	2 migration/0
3 ksoftirqd/0	3 ksoftirqd/0	3 ksoftirqd/0	3 ksoftirqd/0
4 watchdog/0	4 watchdog/0	4 watchdog/0	4 watchdog/0
... 37 lines with identical columns are not shown ...			
32286 bash	32286 bash	32286 bash	32286 bash
32311 HIDEME	32423 ps	32311 HIDEME	32311 HIDEME
32417 ps			

Chapter 6

Conclusion

6.1 Summary of the work

We have presented what honeypots and honeynets are. The current status of available projects with their limitations has been described. We have also presented some of the attacker's techniques that are aimed against honeypots and that help the attacker to hide the used utilities (rootkits).

In Chapter 4 we have made an introduction to the virtualization theory on x86 CPU architecture. Then we have compared currently available solutions and have chosen Xen as the basis for our virtual honeynet.

In Chapter 5 the honeynet built on Xen is described. We have installed and configured the honeypot that mimics a computational cluster. Then we have presented a probabilistic model for simulating users.

The main implementation part of the thesis is also presented in Chapter 5. It is the VAccess library that allows programmer to inspect a virtual machine. We have also presented a GUI application VAmonitor that presents some of the features of VAccess.

6.2 What is missing

We have not implemented any technique that would hide the fact that honeynet is a merely virtual network and runs on virtual machines. We have discussed this problem in Section 4.3 and showed that virtual machines are valid target for attackers and virtual environment is not suspicious (from their point of view) any more.

6.3 Future work

There are many features in our honeynet, that can be enhanced to make the honeynet more convincing and more convenient for an administrator or honeypot developer:

- The simulation model can be enhanced with reactions on failures.

- The simulation model can be enhanced with parameters of commands.
- The VAccess library can be adapted for other VMMs (especially KVM looks very promising lately).
- The VAccess library can hook more system calls like *fork()*, *mmap()* etc.

Appendix A

Breaking out of chroot

In this appendix we describe techniques that allows a user to circumvent chroot jail. By circumventing we mean escaping from a chroot environment and also affecting entities (e.g. processes) outside a chroot environment.

chroot() is not stackable – double chroot When a process calls *chroot()* and then calls it again the kernel does not remember the former root directory and cannot use it to limit the latter *chroot()* call. There is also no limitation to chroot outside of the current root. Although process cannot change its working directory out of **/** it can chroot out of **/**. Kernel does not check the relation between current and requested root. This lets us perform few simple steps to break out of chroot (see figure A.1):

1. Make new temp directory
2. Acquire handle (aka file descriptor) to current root
3. Chroot to the new temp directory
4. Change current working directory up the filesystem tree (this is possible because we are already out of chroot)
5. Chroot to the working directory

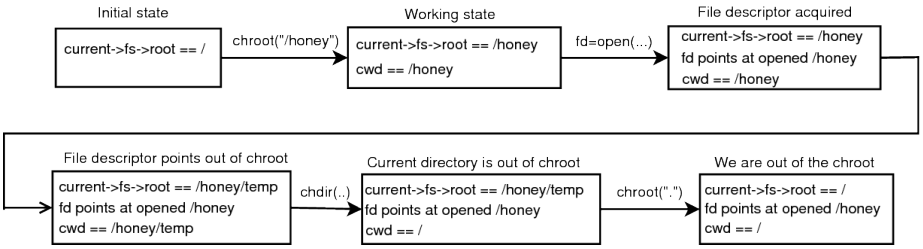


Figure A.1: Breaking out of chroot via “two chroots”

```
# mknod /real_sda2 b 8 2
# mkdir /escape
# mount -t ext3 /real_sda2 /escape
# chroot /escape
```

Figure A.2: Breaking out of chroot via "new root device"

chroot does not care about devices Chroot does not prevent a user from creating any special device. This allows us to create new disk device representing the same disk that is mounted to **/** and mount it wherever we want. Then chroot there and we are out of the chroot. The main problem of this approach is to find out (usually guess) the device mounted as **/**.

Figure A.1 shows a short example of the described technique.

chroot does not limit IPC – shmat(2) Abusing *shmat* system call does not provide the ability to break out of chroot directly. In fact it is a way how a proccess (p1) started in chrooted environment can affect a process outside of chroot (p2).

We assume that process p2 allocates a shared memory. Then p1 can attach to the memory despite the fact it is running in the chroot and virtually at different machine. What the attacker can do then depends heavily on the application.

These facts about shared memory (and other inter process comunication as well) are good proof of the statement that filesystem based limitation is not enough and must be enhanced to be secure.

chroot allows interaction with outside processes – ptrace(2) This system call is even more powerful than *shmat*. With *ptrace* the attacker can force any process running outside of chroot to execute any code on its behalf (see `PTRACE_POKETEXT` in manual page of *ptrace(2)*).

operations in chroot do not check filedescriptors – fchdir(2) As was already mentioned in A kernel does not check whether process running in chroot holds a handle (file descriptor) outside of the chroot. So the double chroot technique can be simplifed if the process already has a filedescriptor for a directory outside of the chroot. In that case the process can change its working directory with *fchdir()* and call *chroot(".")*.

another option for double chroot – pivot_root(2) If the attacker is chrooted and cannot call *chroot()* again (some security patches forbid calling chroot in already chrooted process) she can use *pivot_root()* syscall instead. It is a syscall very similar to *chroot()* and it is used mainly during boot sequence. *pivot_root* is designed to completely migrate **/** of a process. For example script in initrd[1] runs with ramdisk as its **/** and at the end it switches **/** to real root (which is passed to kernel as a parameter).

[1] Initial ramdisk used in linux systems to store modules and scripts. The ramdisk is processed before the root filesystem is mounted.

So *pivot_root()* works almost like *chroot()* and can be used in fchdir technique (see Section A) as the last step.

SUID binaries are always equal – chmod(2) Syscall *chmod()* changes permission bits of a file. We will focus on the `SUID` bit[2]. If the attacker can become a root in chrooted environment and a non-privileged user in real system then she can use the (chrooted) root account to set `SUID` bit to any binary she wants and then run it on behalf of real root by executing the binary in real system under the non-privileged user account.

chroot does not limit access to kernel parameters – _sysctl(2) This approach uses the fact that chrooted environment is not limited in requesting module insertion and that even chrooted superuser (aka root) can modify kernel parameters via *_sysctl()* syscall. The linux kernel has the ability to automaticaly load a module in predefined situations. The insertion is done by executing binary defined in kernel parameter `kernel.modprobe`:

```
# sysctl kernel.modprobe
kernel.modprobe = /sbin/modprobe
```

We can reset this value to our own binary and than persuade kernel to try to load a module (see listing 7). Our malicious binary will be run outside chroot.

unix sockets communication is not limited – abstract unix sockets Some applications use unix sockets for communication between producer and consumer. Sometimes consumer relies on the fact, that kernel allows only privileged process to *connect()* to the socket. Kernel does not check whether privileged user runs in chrooted environment or not. This is a flaw that can be exploited by malicious user and breaks the separation need.

[2]Programmes with `SUID` set run with the priveleges of the owner of the executable not with the privileges of the user who started the program.

Listing 7 Changes kernel parameter and requests module load

```
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/unistd.h>
#include <linux/types.h>
#include <linux/sysctl.h>
#include <stdio.h>
#include <string.h>
_syscall1(int, _sysctl, struct __sysctl_args *, args);
int sysctl(int *name, int nlen, void *oldval,
           size_t *oldlenp, void *newval, size_t newlen)
{
  struct __sysctl_args args={name,nlen,oldval,oldlenp,newval,newlen};
  return _sysctl(&args);
}
#define SIZE(x) sizeof(x)/sizeof(x[0])
#define MODPROBEPATHLEN 256
char modprobepath[MODPROBEPATHLEN];
int pathlen;
int name[] = { CTL_KERN, KERN_MODPROBE };
int main(void){
  strcpy(modprobepath, "/tmp/modprobe_replace");
  pathlen = SIZE(modprobepath);
  if (sysctl(name, SIZE(name), 0, NULL, modprobepath, pathlen))
    perror("sysctl");
  else {
    printf("successfully modified the modprobe path\n");
    socket( AF_SECURITY, SOCK_STREAM, 1);
    printf("executing malicious binary outside chrooted environment\n");
  }
  return 0;
}
```

Appendix B

Installation of a guest system

Guest system is the term used for the operating system running in a virtual machine.

Chroot installation There are various ways how to deploy a new guest. Most modern linux distributions offer a way how to install new instance of the distribution from the running one into a directory mainly for chroot purposes. These are for example:

Gentoo The installation stage-3 method can be used to install gentoo into chroot.

Debian As described in the package debootstrap description: *The utility debootstrap is used to create a Debian base system from scratch, without requiring the availability of dpkg or apt. It does this by downloading .deb files from a mirror site, and carefully unpacking them into a directory which can eventually be chrooted into.*

Scientific linux The package manager yum with option –installroot can be used to install system into a selected directory.

SUSE The configuration tool YaST can be used to install packages to chroot (Yast -> Software -> Installation into directory).

Usually the user creates a file or LVM volume which represents the virtual disk intended to be used by the new virtual machine. After that, he makes the partition layout and creates filesystem on each partition. Then she mounts the partition and use one of the methods above to install distribution.

Another option is to create a file or LVM volume for each partition. This approach is easier to use. There is no need for a virtual boot manager (pygrub, domUloader.py) and mounting the guest partitions in dom0 for maintenance is less difficult.

Direct boot installation All the installation methods mentioned in Appendix B are a bit complicated, because they only install packages and do not configure the system (such as mountpoints, network, kernel and so on). These additional steps can be done manually or there is quite different option.

The other option is to boot Xen with installation kernel and installation ramdisk. This approach is more similar to classic installation. Although the user must still prepare the virtual disk and setup Xen, the installation then lets the user to set up system with particular distribution tools (e.g. system-config- scripts in Scientific Linux).

domi Both approaches can be automated and so is done by project domi[22]. It is a set of shell scripts that allow its user to install debian, fedora, gentoo or suse automatically.

Cloning prepared images Although domi seems to be the right solution, there are some caveats in it. The biggest problem is the installation tools (anaconda in case of Scientific Linux) are not well prepared to run in a paravirtualized environment. It means that the installation gets stuck sometimes during hardware initialization.

The easiest way to setup a new domU is to clone an existing installed image. But where to get the first one to clone? There are several sites and projects where the image can be downloaded or prepared according to user's needs.[1]

We have chosen yet another approach. Complete installation of operating system with minimal set of packages in QEMU[1]. We install the system like we would on a real hardware. Then we mount the disk image and copy all files into an archive called *sl4_minimal.tar.bz2*. This archive is available on the thesis CD.

[1] For example http://cernvm.cern.ch/cernvm/

Appendix C

Details of kernel memory modifications

This appendix describes details of kernel memory modifications done via `/dev/kmem` device.

Accessing kernel memory For modifying running kernel we can use `/dev/kmem` which is a device that provides image of current kernel virtual memory. So, if we know address of some structure in kernel memory we can modify this structure simply by rewriting appropriate offset in this device[1].

Finding syscall table Typical rootkit needs to modify syscall table. When LKM is disabled in kernel, the attacker has no information about the address of the syscall table. This address must be first determined before the syscall can be changed.

On i386 architecture we first get the address of Interrupt Descriptor Table[2]. Line 0x80 of the table defines address of handler for all system calls. This handler calculates the address of requested system call and jumps to the address. The address of the syscall table is used for the calculation and the code can be found in kernel source in `arch/i386/kernel/entry.S`.

We have to go through the system call handler and look for op-codes of instruction *call someoffset(,%eax,4)*. The *someoffset* is then the address we are looking for – *sys_call_table*.

Finding kmalloc() When we need to execute some code in kernel context we first need to have some memory where we can insert this code. The function that allocates memory for kernel is called *kmalloc()* or we can use more low level *__kmalloc()*[3].

[1] We use the UNIX paradigm that (almost) every device is a file.

[2] This is done with instruction *sidt*.

[3] Actually, we have to use __kmalloc() because kmalloc() is an in-line function so we cannot find it as a standalone function and call it.

But where cat we get the address of the function when the kernel does not support modules (and so it does not export symbols and their addresses in `/proc/kallsyms`)?

The idea presented in [39] is to go through kernel memory and find all occurrences of function call with two parameters[4]. Then, we assume that the *kmalloc()* is the most often called function. So, the last step is to get the most often argument of instruction *call* and use it as the address of *kmalloc()* function. The example code that counts occurrences of given type of function call, compares it with `/proc/kallsyms` and proves the previous statement can be found on enclosed CD in `sources/kmalloc_locator.py`. For example, on vanilla kernel 2.6.22.1 the output for first 14 MB looks like in listing 8.

Listing 8 Output of kmalloc_locator.py for first 4MB of memory

```
### Trying to get the __kmalloc from /proc/kallsyms
__kmalloc: c047003a
### Analyzing first 14MB of memory
Address                 count   Name (/proc/kallsyms)
-----------------------------------------------------
c047003a:                  22     __kmalloc
c045e7a5:                  12     __kzalloc
c044c3a9:                   6     audit_log_start
c044ab58:                   4     __cpuset_zone_allowed_hardwall
c04dc581:                   3     kobject_get_path
c045e7dd:                   3     kstrdup
c049c0b0:                   2     posix_acl_clone
c04582f1:                   2     try_to_release_page
c046fb05:                   2     kmem_cache_zalloc
c0599784:                   1     skb_clone
c04493c6:                   1     kimage_alloc_page
83e0152e:                   1     X
c044610f:                   1     memory_bm_create
c049c0d3:                   1     posix_acl_alloc
c05cd275:                   1     tcp_send_active_reset
c0561269:                   1     usb_alloc_urb
c0471ab9:                   1     __percpu_alloc_mask
c04dc3da:                   1     idr_pre_get
```

Invoking kmalloc Now when we have address of *__kmalloc* function we need to call it. Of course we cannot call it directly from user space[5]. Instead, we have to trick kernel to call the function for us.

This can be done by rewriting the code of some syscall with our own routine through `/dev/kmem` and issuing the system call from user space. The code should allocate given amount of memory and return its address. The exact steps are:

1. Write position independent code that just calls *__kmalloc()* and returns the address of the memory.
2. Compile it into binary (not object binary file, just a binary string with instructions).
3. Find *sys_call_table* address.

[4] We can even be more precise and get only functions whose one parameter is GFP_KERNEL. It is a constant used as a parameter of __kmalloc() very often.

[5] Accessing kernel address space from user program results in SIGSEGV signal for current process.

4. Get the address of a rarely used system call (e.g. *kexec_load*) from *sys_call_table* and store its code for step 8.
5. Replace the code of the system call (by writing to `/dev/kmem`) chosen in step 3 with the code compiled in step 1.
6. Issue the system call from user space and get the returned address.
7. Write the malicious code to `/dev/kmem` on address from step 6.
8. Replace back the code replaced in step 4.

The malicious code mentioned in step 7 is typically a different implementation of system call that can hide processes or files owned by the attacker.

Bibliography

[1] Fabrice Bellard. Bochs homepage. http://fabrice.bellard.free.fr/qemu.

[2] Jasmin Blanchette, Mark Summerfield, Jamie Cameron, Mel Gorman, Christopher R. Hertel, Anthony J. Massa, Nigel Mcfarlane, Rafeeq Ur Rehman, Christopher Paul, Rafeeq Ur Rehman, John H. Terpstra, Jelmer R. Vernooij, John H. Terpstra, Mel Gorman, and Prentice Hall. Understanding the linux ÂⓇ virtual memory manager.

[3] Silvio Cesare. Runtime kernel kmem patching. http://vx.netlux.org/lib/vsc07.html, 1998.

[4] Ramkumar Chinchani, Aarthie Muthukrishnan, Madhusudhanan Chandrasekaran, and Shambhu Upadhyaya. Racoon: Rapidly generating user command data for anomaly detection from customizable templates. *acsac*, 0:189–204, 2004.

[5] Intel corporation. Intel virtualization technology specification for the ia-32 intel architecture. ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf, April 2005.

[6] Jeff Dike. User mode linux homepage. http://user-mode-linux.sourceforge.net/.

[7] Maximillian Dornseif, Thorsten Holz, and Christian Klein. Nosebreak - attacking honeynets. In *Proceedings of the 2004 IEEE Information Assurance Workshop*.

[8] Mike Ciavarella et al. The jail aware scheduling project. http://www.cs.mu.oz.au/jas/, 2004.

[9] Peter Rundberg et al. Freebench. http://www.freebench.org/, 2002.

[10] FuSyS. Kstat – kernel security therapy anti-trolls. http://s0ftpj.org/tools/kstat24_v1.1-2.tgz.

[11] Jacques Gelinas. Virtual private servers and security contexts. http://www.solucorp.qc.ca/miscprj/s_context.hc?dp=0&full=1&prjstate=1&nodoc=0, 2004.

[12] grsecurity team. grsecurity. http://www.grsecurity.net/index.php.

[13] grsecurity team. grsecurity chroot – comparison. http://grsec.linux-kernel.at/compare.php.

[14] "grugq". The design and implementation of userland exec. http://lists.grok.org.uk/pipermail/full-disclosure/attachments/20040101/fea4fb1f/ul_exec.txt, 2004.

[15] "grugq" and "scut". Armouring the elf: Binary encryption on the unix platform. http://www.phrack.org/issues.html?issue=58\&id=5\#article, 2001.

[16] Daniele Gunetti and Giancarlo Ruffo. Intrusion detection through behavioral data. In *IDA '99: Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis*, pages 383–394, London, UK, 1999. Springer-Verlag.

[17] Thorsten Holz and Frederic Raynal. Detecting honeypots and other suspicious environments. In *Proceeedings of the 6th IEEE Information Assurance Workshop*. IEEE, 2005.

[18] Tripwire Inc. Tripwire. http://www.tripwire.com/, 2006.

[19] VMware Inc. VMware homepage. http://www.vmware.com/.

[20] Ludek Matyska Jiri Denemark, Miroslav Ruda. Virtualizing metacentrum resources using magrathea. *Networking Studies II: Selected Technical Reports*, 1:129–145, 2008.

[21] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. http://docs.freebsd.org/44doc/papers/jail/jail.ascii.gz, 2000.

[22] Gerd Knorr. domi. http://linux.bytesex.org/, 2006.

[23] Neel Mehta and Shaun Clowes. Secureреality. http://www.secureреality.com.au/, 2003.

[24] Sun microsystems. Containers (zones). http://www.sun.com/bigadmin/content/zones/.

[25] Nelson Murilo. chkrootkit. http://www.chkrootkit.org/, 2006.

[26] European Network of Affined Honeypots. D0.1: Survey on the state-of-the-art. http://www.fp6-noah.org/publications/deliverables/D0.1.pdf, 2005.

[27] Parallels. Openvz homepage. http://openvz.org/.

[28] Parallels. Virtuozzo homepage. http://www.virtuozzo.org/.

[29] Petroni, Fraser, Molina, and Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *13th USENIX Security Symposium*, pages 179–194.

[30] PhysTech. Ubench homepage. http://www.phystech.com/download/ubench.html, 1998.

[31] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Communications of the ACM*, July 1974.

[32] Herbert Potzl. Linux-vserver white paper. http://linux-vserver.org/index.php?page=Linux-VServer-Paper, 2004.

[33] The Honeynet Project. Know your enemy: Sebek. http://www.honeynet.org/papers/sebek.pdf, 2003.

[34] Danny Quist and Val Smith. Detecting the presence of virtual machines using the local data table. http://www.offensivecomputing.net/files/active/0/vm.pdf.

[35] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. http://citeseer.ist.psu.edu/robin00analysis.html, 2000.

[36] J. S. Robin. Analyzing the intel pentium's capability to support a secure virtual machine monitor., 1999.

[37] Neil C. Rowe. An intrusion-detection environment for information-security instruction. *29th ASEE/IEEE Frontiers in Education Conference*, 0:13a9–6, 1999.

[38] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models for automated test case generation. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[39] "sd" and "devik". Linux on-the-fly kernel patching without lkm. http://www.phrack.org/phrack/58/p58-0x07, 2001.

[40] Amit Singh. An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/.

[41] Noah Spurrier. Pexpect. http://www.noah.org/wiki/Pexpect, 2008.

[42] Inc. Sun Microsystem. System administration guide: Solaris containers-resource management and solaris zones. http://docs.sun.com/app/docs/doc/817-1592, 2005.

[43] Bochs team. Bochs homepage. http://bochs.sourceforge.net/.

[44] Plex86 team. Plex86 homepage. http://www.plex86.org/.

[45] Xen team. Xen homepage. http://www.xensource.com/xen/.

[46] Rich Uhlig and Larry Smith et al. Intel virtualization technology. http://cache-www.intel.com/cd/00/00/22/19/221961_221961.pdf, May 2005.

[47] various authors. SentinelSecurityToolkit. http://zurk.sourceforge.net/zfile.html, 2001.

[48] Virtualization.info. Glossary. http://www.virtualization.info/glossary/#E, 2008.

[49] Linux vserver team. Linux vserver homepage. http://linux-vserver.org/.

[50] Wikipedia. Freebsd jail — wikipedia, the free encyclopedia, 2006. [Online; accessed 19-May-2006].

[51] Wikipedia. Ibm power — wikipedia, the free encyclopedia, 2006. [Online; accessed 19-May-2006].

[52] Wikipedia. Rootkit — wikipedia, the free encyclopedia, 2007. [Online; accessed 12-September-2007].

[53] "zeppoo". zeppoo. `http://sourceforge.net/projects/zeppoo`, 2006.